INITIALIZATION DESIGN FOR DYNAMIC DETERMINATION OF RESOURCES

Gary Stewart Baker

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

INITIALIZATION DESIGN FOR
DYNAMIC DETERMINATION OF RESOURCES

by

Gary Stewart Baker

June 1981

Thesis Advisor: R. R. SCHELL

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Initialization Design for Dynamic Determination of Resources		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis: June 1981 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Gary Stewart Baker		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1981 13. NUMBER OF PAGES 109
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Initialization, dynamic determination of resources, multi-microprocessor, initialization design, secure archival storage system, information security

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis presents a versatile initialization design for dynamic determination of physical resources in an adaptive manner for a multi-microprocessor environment. The design is general in nature and represents a structured, functional approach to the initialization process based on the use of dynamic resource mapping, knowledge passing between layered program components, and coordinated interprocessor communication. An implementation of this design is presented for initialization of the Secure Archival Storage System. The hardware architecture utilizes the commercially available,

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

8000 based Advanced Micro Computer Am96/4116 MonoBoard Computer, configured
o support information security.

Approved for public release; distribution unlimited.

Initialization Design for Dynamic Determination of Resources

by

Gary Stewart Baker
Lieutenant Commander, United States Naval Reserve
B.S., California State College, Los Angeles, 1970

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1981

ABSTRACT

This thesis presents a versatile initialization design for dynamic determination of physical resources in an adaptive manner for a multi-microprocessor environment. The design is general in nature and represents a structured, functional approach to the initialization process based on the use of dynamic resource mapping, knowledge passing between layered program components, and coordinated interprocessor communication. An implementation of this design is presented for initialization of the Secure Archival Storage System. The hardware architecture utilizes the commercially available, Z8000 based Advanced Micro Computer Am96/4116 MonoBoard Computer, configured to support information security.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my appreciation to my thesis advisor, Colonel Roger R. Schell, USAF, for his enthusiastic guidance and support. Without the many hours of counseling and endless supply of ideas he provided, this thesis could not have been completed.

My thanks also go out to Professor Uno R. Kodres for the support and advise he offered in the early stages of the hardware efforts.

A special thanks and mention is due Mr. Bob McDonnell and Mr. Mike Williams of the Computer Science Laboratory, whose tutoring and assistance in hardware-related areas was invaluable during my research effort. Their capacity for patience is unsurpassed.

I am also appreciative of the helpful comments given by my reader, Professor Dusan Badal.

Most of all, I want to express my sincere thanks to my family, my wife Chris, and sons Gary and Cameron, for their understanding and encouragement, and for the sacrifices they have made during my graduate studies.

I. INTRODUCTION

Initialization is the recurring process of effecting a normally running operating system on a given hardware configuration each time the system is started. This thesis presents a versatile initialization design for general application, which dynamically determines the system configuration in an adaptive manner. The initialization design is implemented specifically for a member of a family of secure, distributed, multiprogramming, multi-processor operating systems. The Secure Archival Storage System (SASS) is that member. A Zilog Z8000 microprocessor-based hardware architecture was developed that will support ongoing SASS development, and is used in this design implementation as the hardware base.

The key features of this initialization design include its versatility and general applicability, its use of dynamic resource mapping, its adaptive use of dynamically defined resources, and its hardware synchronization methods. The versatility is derived from use of knowledge passing rather than assumption between components. Dynamic resource mapping is the defining of hardware resources, (in particular, processors and primary storage) as the initialization process progresses. The adaptive use of hardware resources is facilitated by the ability to

dynamically determine where to locate programs and data in primary storage. The hardware synchronization method makes use of a shared data structure to facilitate inter-processor communication and coordination.

A. MOTIVATION

O'Connell and Richardson[1] outlined a high level design for a family of secure distributed, multi-processor operating systems, with the primary motivation of (1) effectively coordinating the processing power of microprocessors and (2) providing information security. A subset of this family, the Secure Archival Storage System (SASS) [2,4], has been selected as a testbed for the general design. SASS will provide consolidated file storage for a network of possibly dissimilar "host" computers. The system will provide controlled, shared access to multiple levels of sensitive information [5]. A complete description of this family of operating systems and the SASS can be found in reports prepared by Schell and Cox [25,26]. The hardware was selected and a development system based on the hardware was procurred. A Zilog Z8000 microprocessor based Developmental Module [24] integrated with a Z80 microprocessor based developmental system has been the instrument of continuing SASS research. Further efforts by Rietz[5], Coleman[3], Wells[6], and Strickler[7] have brought the development to the point to where there is a need for a multiprocessor

environment. The Developmental Module does not support multiprocessor development.

To support concurrent work on the SASS and to act as a testing platform for the initialization design proposed in this thesis, the SASS Developmental Architecture was designed and built. At the foundation of the SASS Developmental Architecture is the commercially available Advance Micro Computer Am96/4116 MonoBoard Computer with a standard Multibus (INTEL) interface. In this application the intent was to match the hardware architecture to the needs of the operating system design, rather than the more common approach of building an operating system around available hardware features.

B. BACKGROUND

Two important key concepts have had an underlying influence on this thesis. These concepts are machine virtualization and dynamic reconfiguration. It is essential that these be understood before proceding into a discussion of initialization design.

The interfacing of hardware and software is a subject which has not adequately been covered in the literature. In fact, the subject of initialization as a whole can not be referenced to any significant degree. This interfacing between the "bare machine" and the software environment is called the basic machine interface. The basic machine

interface consists of the set of all the software visible objects and instructions that are directly supported by the hardware and firmware of a particular system. A recent work by Ross[18] has produced a simple, flexible initialization design that establishes an environment for a basic machine interface. This is not supported directly on a bare machine but is instead supported in a manner similar to an extended machine interface. Extended machine interfaces have long been used to allow application programs to run on different machines; they commonly take the form of interpretors. This basic machine interface is known as a virtual machine and is a basic feature of the SASS. Ross' initialization design builds from the "bare machine" to this virtual machine environment using a layered approach as described by Luniewski[17], to establish the interfacing.

The "bare machine" refered to in most initialization designs is in some manner assumed. Luniewski in his design assumed a minimal hardware configuration for the initialization mechanism. Given this minimal hardware configuration (which he defines to be a subset of the largest potential hardware configuration) his initialization mechanism employed dynamic reconfiguration to establish the actual "bare machine" or hardware configuration. Dynamic reconfiguration is the changing of the system configuration while the system is running. From a hardware configuration

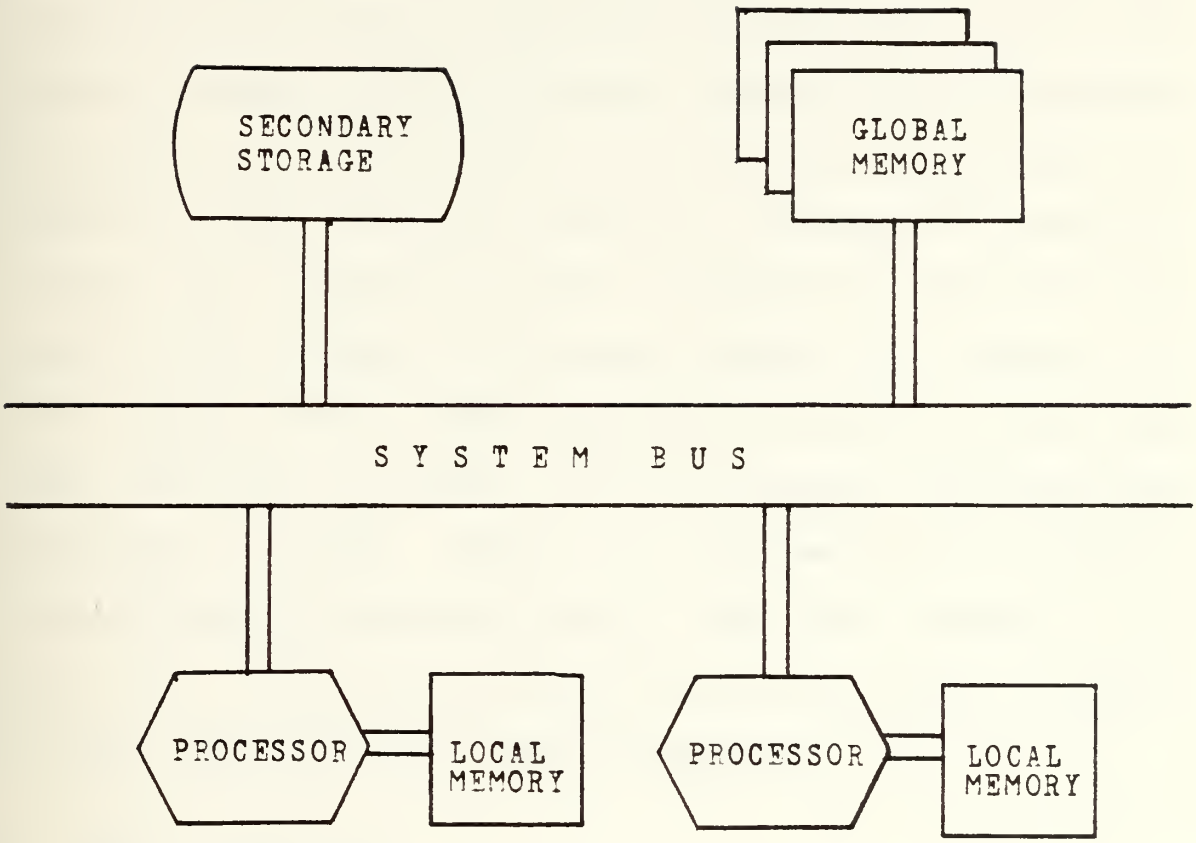
viewpoint, this means the dynamic addition or subtraction of known hardware resources.

Various hardware resources referred to in this thesis are shown in Figure I-1. A multiprocessor environment refers to having more than one processor interconnected with a system bus. Local memory is that primary memory accessed by only one processor; and global memory is that primary memory addressable by all processors interfacing to the system bus. Data storage interfaced to the system bus having access controlled by a separate processor, is defined as secondary storage.

C. OBJECTIVES

The objectives of this thesis are threefold: (1) to show the need for a method of dynamic determination of hardware resources, (2) to present an initialization design to illustrate the methodology, and (3) to actually implement the design with the SASS and its hardware architecture to show it is practical. An additional conscious effort was made to include within this thesis all documentation concerning the hardware and software tools necessary to duplicate the results. Extensive use was made of the appendices for this purpose.

Reference [27] presents a summary description of the hardware for the SASS. It contains detailed wiring configurations for both the MonoBoards and Ram Memory Boards



Multiprocessor Environment
Figure I-1

'Am96/1000), including detailed description of the wiring modification for information security of the MonoBoard local memory. It also contains the SASS Developmental Monitor program listing with a command syntax tutorial, the Bootload program listing for the firmware, and the Bootstrap program listing for use with the SASS. The bootstrap program can be adapted to other operating systems by changing the manner of loading from secondary storage. In addition, reference [27] contains the listings for programs written in support of this thesis effort. These programs were used to effect the object code file transfer to the Intellec Microcomputer Development System (MDS) for programming the EPROM's (electronically programmable ROM). The same program also served for transferring the source code listings contained in these appendices, for text processing.

D. THESIS STRUCTURE

In this chapter a brief discussion was presented on the motivations and influences brought to bear on this thesis effort. The objectives were stated and the documentation goals were established. Chapter II addresses the subject of initialization and proposes an initialization design methodology. Chapter III presents the SASS and its hardware configuration to allow for a complete environment definition. This environment served as a model for exploring issues pertaining to initialization design. Chapter IV

describes the implementation for the actual hardware architecture. The final chapter presents conclusions and observations that resulted from this thesis effort and suggestions for further research.

II. INITIALIZATION DESIGN

The objective of system initialization is to get an operating system loaded and running on a computer system. This task must be accomplished each time a computer system is powered-up or a change to a new or revised operating system is made. In the past, this has been considered an implementation detail, specific to a given system. As a result, existing system initialization schemes are not of a general nature or structured in design. This chapter will examine system initialization from the standpoint of a time ordered sequencing of activities and functional grouping of tasks.

A. INITIALIZATION CONCEPTS

The general form of system initialization is to have a bootload medium, that contains the necessary programs and data, bring in the operating system from some external storage device and effect normal operations. The manner in which this is accomplished can be identified as occurring in three time ordered phases: system generation phase, initialization phase, and run time phase [17]. System generation time is that phase where in the bootload medium and operating system core images are generated (created). This usually occurs in the same environment in which the

operating system is developed. Initialization time is that period of time from initial power-up of the computer system to the point where the operating system is running normally. The time after the system is initialized, when it is running normally is called the run time phase.

Luniewski[17] proposed a simple and versatile mechanism for system initialization based on these three phases and the underlying premise that an activity performed at system generation time or at run time is inherently simpler than the same action performed at initialization time. His method is based on the idea of a layering of functions begining with an assumed minimal configuration, and the concept of dynamic reconfiguration [11] to develop the system on which the core image of the operating system will run. The minimal configuration, which is a subset of resources contained in the full hardware configuration, was assumed to have a single processor, a given primary memory of known size and known physical address, and system tables of known size. These assumptions allowed for a significant amount of initialization tasks to be performed during the system generation phase and run time initialization (viz., specific resource virtualizations).

These initialization concepts were employed in subsequent work by Ross[18] to design an initialization mechanism for a particular real-time application. His approach defined the initialization phase as a "two load"

operation where a ROM resident (in hardware) bootload program loads a core image of the bootstrap program which procedes to load and develop the environment for the full operating system core image. This concept of a bootload program and bootstrap program is driven by the desire to keep the bootload program (in ROM) small for hardware considerations.

Each of these initialization programs (i.e., bootload, bootstrap, operating system core image) are statically provided with information about the others. The bootload program knows at what physical address to load the bootstrap program based on the minimal configuration assumed. The bootstrap program knows at what physical address to load the operating system core image based on the restrictions caused by absolute addressing contained in the operating system code; in addition the bootstrap program makes fixed physical resource allocations based on a knowledge of operating system core image system tables. The operating system knows the processor(s) data structures (i.e., stacks, PSA) from information statically provided at system generation time.

A more versatile initialization design would be to take a dynamic approach to information passing between initialization programs. This requires a dynamic determination of the environment in an adaptive manner; knowledge of real resources is gained experimentally, with the new knowledge then used to gain further definition. The

total resource knowledge would then be passed to the resource managers. In this manner a layered real environment would build to the base of the layered virtual environment refered to in the above works.

B. SYSTEM GENERATION PHASE

The system generation time includes all manner of software development for system initialization when considered in the broadest terms. A more restricted and classical view proposes generation of the operating system core image and perhaps a portion of the bootstrap program. In any case the software developed specifically and of necessity for the operating system of concern is produced during this phase. As discussed earlier, as many initialization tasks as possible should be allocated to this phase and the run time phase. During the system generation phase the production environment is more hospitable, having full operating system services available; a similar environment exists at the begining of run time. The initialization phase on the other hand, has only the bare hardware and what it can built from this to run on.

The software used during the initialization process is composed of the bootload program, the (possibly several) bootstrap program, and the operating system core image. All are necessary to effect initialization in a "two load" environment; however, not all are required to be produced

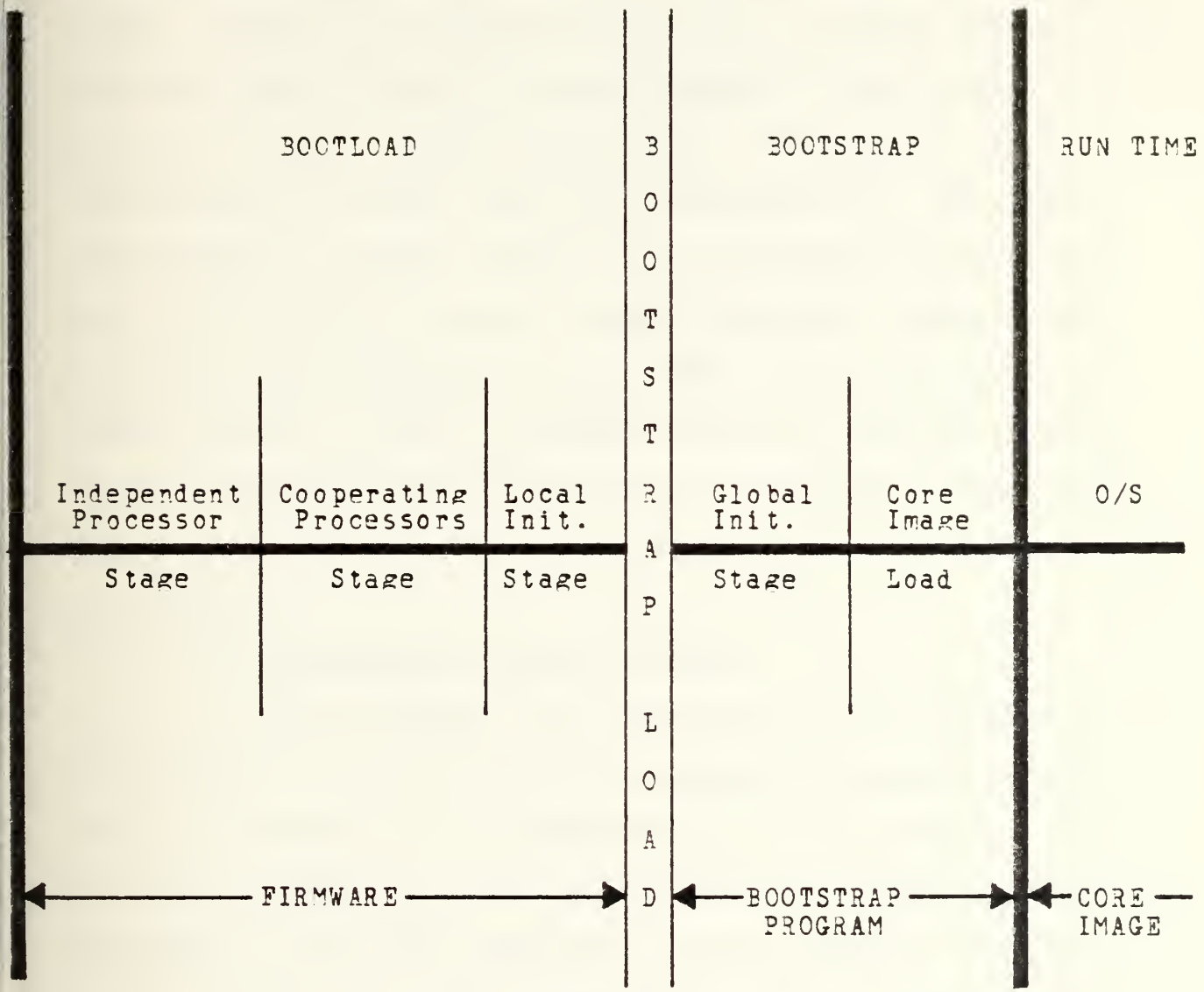
during the same system generation phase. Using the proposed adaptive method for dynamic determination of physical resources should allow some degree of portability for the programs. For instance, the bootload program can dynamically determine where to load the bootstrap programs; and the bootstrap program(s) can in turn do the same with the operating system core image. This would mean that the bootload program could service a variety of different bootstrap programs; and a single bootstrap program could support a variety of bootstrap loader programs, each produced specifically for loading a given operating system.

One restriction on the program development must exist during the system generation phase to support the proposed method. In the absence of a linking loader in the initialization code, which has been dismissed adequately by Luniewski[17] as impractical, program development must proceed in an environment free of absolute addressing. This will insure that, regardless of where the program's core images are positioned in physical memory, code execution can be started and will continue to completion. In the case of the ROM resident bootload program, its application will not be limited to a specific hardware architecture; and the bootstrap programs and operating system core image will have the same degree of configuration independence.

C. INITIALIZATION PHASE

The initialization phase begins at the moment a hardware "bootload" signal is applied to a computer system and ends with an operating system running normally. As stated above the participants of this phase are a bootload program, one or more bootstrap programs, and the operating system core image. These elements can serve as natural partitions for a time sequencing of the bootload phase. Execution starts in the bootload program, control is passed to the next sequencial bootstrap program, and so forth. Once the control flow has left a given partition, it does not return. The only exception might be the case of a return to a monitor based program for a developmental system, which most likely will be ROM resident. This monitor program, however must be considered a separate program from the bootload program.

Within each of the time sequencing partitions the tasks performed can be grouped into stages for ease of definition. These stages are not necessarily disjoint in time; some parts of the tasks may be performed concurrently. This initialization phase organization is shown in Figure II-1. Bootload operations occur in three stages: (1) independent processor stage, (2) cooperating processor stage, and (3) local initialization stage. The bootstrap program operations are divided into the global initialization stage and the core image load stage.



Initialization Phase
FIGURE II-1

1. Bootload Operations

Those initialization tasks performed in the bootload program occur in three stages: the independent processor stage, which is characterized by a single processor awareness and a variable free environment; the cooperating processor stage, which begins with the first use of a multiprocessor mutual exclusion mechanism to facilitate multiprocessor shared memory in a coordinated fashion; and local initialization stage in which the local software and hardware initializations occur. The bootload program, as stated earlier, is ROM (or EPROM) resident; this realization of the bootload program in hardware is commonly called the firmware and will be termed thus throughout the remainder of this thesis.

a. Independent Processor Stage

The begining of the initialization process is defined by an initial execution point and an initial address space (in keeping with the definition of a process). The processor itself has only limited internal resources and no knowledge of any other processors. Its internal resources consist solely of its register structure. The initial execution point is obtained by the processor from an address location defined internally within the processor. Commonly, on power-up or reset all internal registers are cleared (zeroed) and the instruction counter then used to 'fetch' the initial execution point. In the case of the Z8000, for

example, physical address zero would contain the initial execution point; in our implementation one address (0002 HEX) would contain the processor status (FCW) and another (0004 HEX), the program counter (PC) value. The initial address space as known by the processor is the firmware.

The goal of the independent processor stage is to dynamically determine the existence of physical memory resources; knowledge as to the size and addresses must be obtained. Without this knowledge, the processor is working in a variable free environment, using only its registers for data storage.

This dynamic function of memory determination is accomplished in three tasks: clearing of memory, defining memory, and mapping of memory. Figure II-2 presents pseudo-coding for each of these functions. The 'clearing' of memory requires that portions of memory being mapped, be brought to a known state, which removes the probabilistic aspects of the task. Conventional notions of a memory map divides the memory into blocks. These blocks will be sampled during the mapping to dynamically determine if they exist. The location for sampling within each block must be brought to this known state. Defining memory then becomes a task of checking for read/write capability of each memory block. Once the existing locations are found, it is possible to build the entire memory map, during the memory mapping task.

```
CLEAR_MEMORY:
   BLOCK_ADR := START_BLOCK
   SCRIBE_ADR := BLOCK_ADR + 1
   DO
      @BLOCK_ADR := R/W_PATTERN
      @SCRIBE_ADR := Ø
      BLOCK_ADR := BLOCK_ADR + BLOCK_SIZE
      SCRIBE_ADR := BLOCK_ADR + 1
   UNTIL (BLOCK_ADR > END_BLOCK)
   WAIT

SCRIBE_MEMORY:
   BLOCK_ADR := START_ADR
   SCRIBE_ADR := BLOCK_ADR + 1
   DO
      IF (@BLOCK_ADR = R/W_PATTERN) THEN
            LOCK_SYSTEM_BUS
            @SCRIBE_ADR := @SCRIBE_ADR + 1
            UNLOCK_SYSTEM_BUS
      BLOCK_ADR := BLOCK_ADR + BLOCK_SIZE
      SCRIBE_ADR := BLOCK_ADR + 1
   UNTIL (BLOCK_ADR > END_BLOCK)
   WAIT

DEFINE_MEMORY:
   LOW_GLOBAL_BLOCK := NON-EXISTENT
   LOW_LOCAL_BLOCK := NON-EXISTENT
   CPU_COUNT := 1
   BLOCK_ADR := END_BLOCK
   SCRIBE_ADR := BLOCK_ADR + 1
   DO
      IF (@BLOCK_ADR = R/W_PATTERN) THEN
            IF (@SCRIBE_ADR = 1) THEN
               LOW_LOCAL_BLOCK := BLOCK_ADR
            ELSE IF (@SCRIBE_ADR >= CPU_COUNT) THEN
                  CPU_COUNT := @SCRIBE_ADR
                  LOW_GLOBAL_BLOCK := BLOCK_ADR
            BLOCK_ADR := BLOCK_ADR - BLOCK_SIZE
            SCRIBE_ADR := BLOCK_ADR + 1
      ELSE
            BLOCK_ADR := BLOCK_ADR - BLOCK_SIZE
            SCRIBE_ADR := BLOCK_ADR + 1
   UNTIL (BLOCK_ADR < START_ADR)
   RETURN (LOW_LOCAL_BLOCK, LOW_GLOBAL_BLOCK,
           CPU-COUNT)
```

Independent Processor Stage
Figure II-2

A sample location is brought to the known state by writing to it a specific pattern. This pattern is used to check the operational status of the hardware memory devices (RAM). Each device must be able to contain data in both states, i.e., 1 and 0; therefore a write-then-read operation should be performed with 1's and Ø's. In a byte oriented memory organization, for example, RAM may consist of eight devices, each contributing one bit for each byte of data. In such a case, the pattern would be written and read from one byte (e.g., '55') and the inverse pattern (e.g., 'AA') written and read to another. Using this example in a 16-bit architecture would require a one word ('55AA') read/write operation to verify the operational status of a block of the byte oriented memory organization.

The size of the blocks that divide the physical memory space is determined from consideration of the hardware organization of memory, the sizes of any natural partitions (i.e., ROM), and in certain cases the aspects of memory virtualization to be used later. In our implementation, for example, the minimum size of the blocks was determined by the minimum ROM size that the processor architecture could support (2K bytes). The entire physical address space is then divided into blocks of this size, and an appropriate location within the block (usually the first address) designated for sampling. The individual physical memory devices are checked by use of the read/write pattern

in the manner described above. Once a device is found operational and accessible for one address, it can be assumed the same for all addresses within the block since each address also enables the device. The entire physical address space, taken in blocks, is then cleared, defined and mapped.

Included with the code contained in firmware is an identifier to be used by the processor for its own unique identification within the system. This value can be put into the firmware by serializing each ROM as it is programmed with the bootload program. At the end of the independent processor stage the processor has knowledge of its own memory space (RAM) in the form of a memory map; its own unique processor identification obtained from serialized firmware; and the location of its firmware within the address space.

b. Cooperating Processor Stage

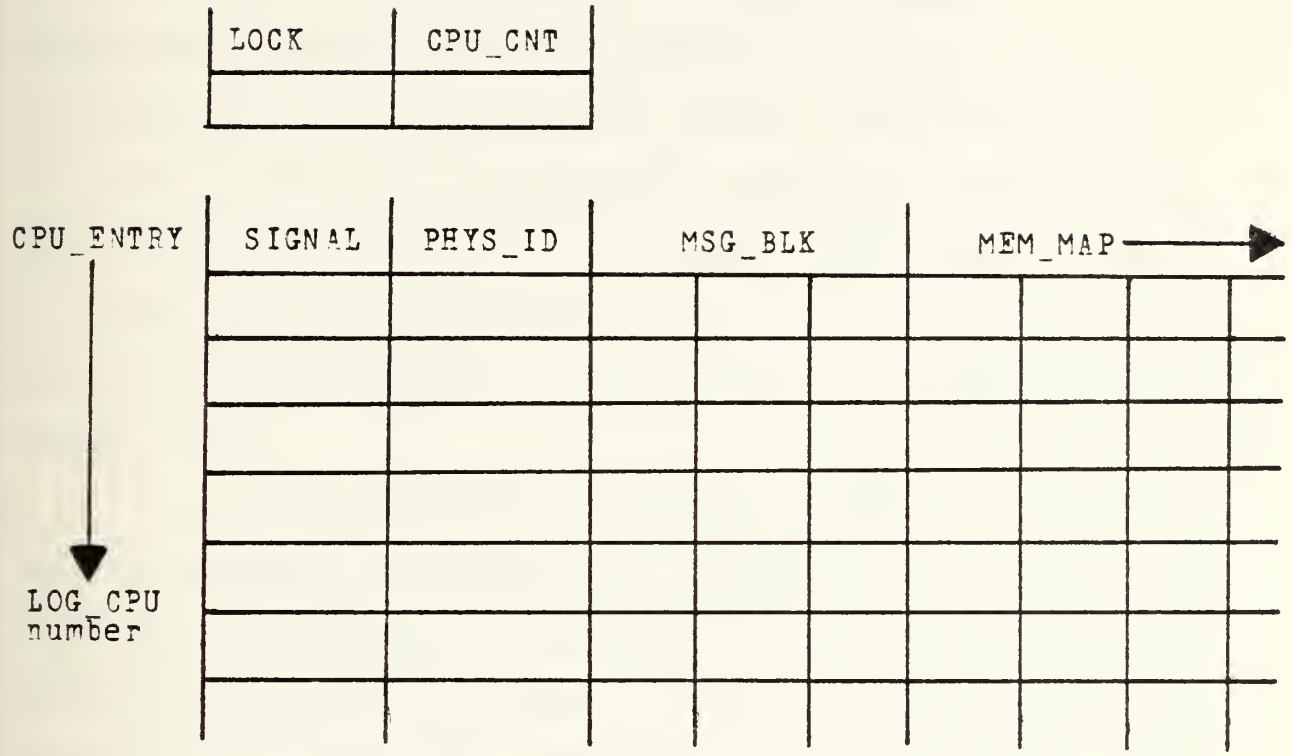
The cooperating processor stage begins with the assumed existence of other processors, which requires the use of a mutual exclusion mechanism to preserve the integrity of a shared memory. Some "read-alter-write" operation (e.g., a test and set instruction) is needed. A typical read-alter-write operation involves fetching a value from memory, modifying the value, and writing it back.

The objectives of this stage are to define the multiprocessor environment and to select a "bootload processor" to coordinate the loading of, and transferring of control between bootstrap and operating system core image programs. A single processor for interfacing with secondary storage is necessary to prevent the interference that would result from multiple commands to secondary storage. The addresses to be loaded from secondary storage must be determined by only one processor. Definition of the multiprocessor environment requires determining the total number of processors, their cooresponding unique identifications, and the existence of any shared memory. Each of these is facilitated by a method of having each processor "make itself known" to the multiprocessor system; this is done by scribing memory.

Scribing is the operation in which each processor uses the read-alter-write operation with mutual exclusion, to increment by one (make itself known) the value of a pre-defined location within each accessible block of memory. For the Am96/4116 MBC mutual exclusion is provided by "locking" the system bus to prevent other processors from interferring with the operation. After all processors have completed scribing their own address space, the highest value found at this location from all memory blocks is the total number of system processors which are defined to have global memory. Global memory is composed of those memory

blocks having the maximum number of processors scribed into them. Those memory blocks having less than the maximum but greater than one processor (local memory) are not considered local or global. They cannot function as local memory because of the possibility of information overwrite; or as global memory because they are are not accessible by all. Use of this memory becomes an implementation detail. The consolidation of all processors' memory maps into a system memory map and the assignment of logical CPU numbers (system use) to physical processors (unique ID), would complete the multiprocessor environment definition.

To coordinate the multiprocessor efforts in defining this system environment, a bootload processor, called the bootload CPU, must be established. The bootload CPU coordinates the activities of the other processors by use of a system data structure known as the Configuration Table. This structure, as seen in Figure II-3, is initially used to determine the Bootload CPU. The table is implicitly established at an available global memory address, and a deliberate race condition is effected to select the Bootload CPU. Each processor attempts to gain exclusive access to the configuration table by successfully setting the table lock; a test and set operation must be performed to set this lock. The use of the table lock insures sequential access to the table.



Configuration Table
Figure II-3

To insure the integrity of this test and set operation in a multiprocessor environment, a mechanism to lock the system bus must be employed. The locking of the bus prevents other processors from using it. An intentional race condition is established whereby each processor attempts to lock the system bus so as to gain access to the configuration table. This race condition can then be used to differentiate the processors and thereby select the bootload CPU. For instance, in the algorithm depicted in Figure II-4, each processor obtains its own logical CPU number on entry into the table and determines if it is the bootload CPU. The bootload CPU has logical CPU number Ø; all others become member CPU's.

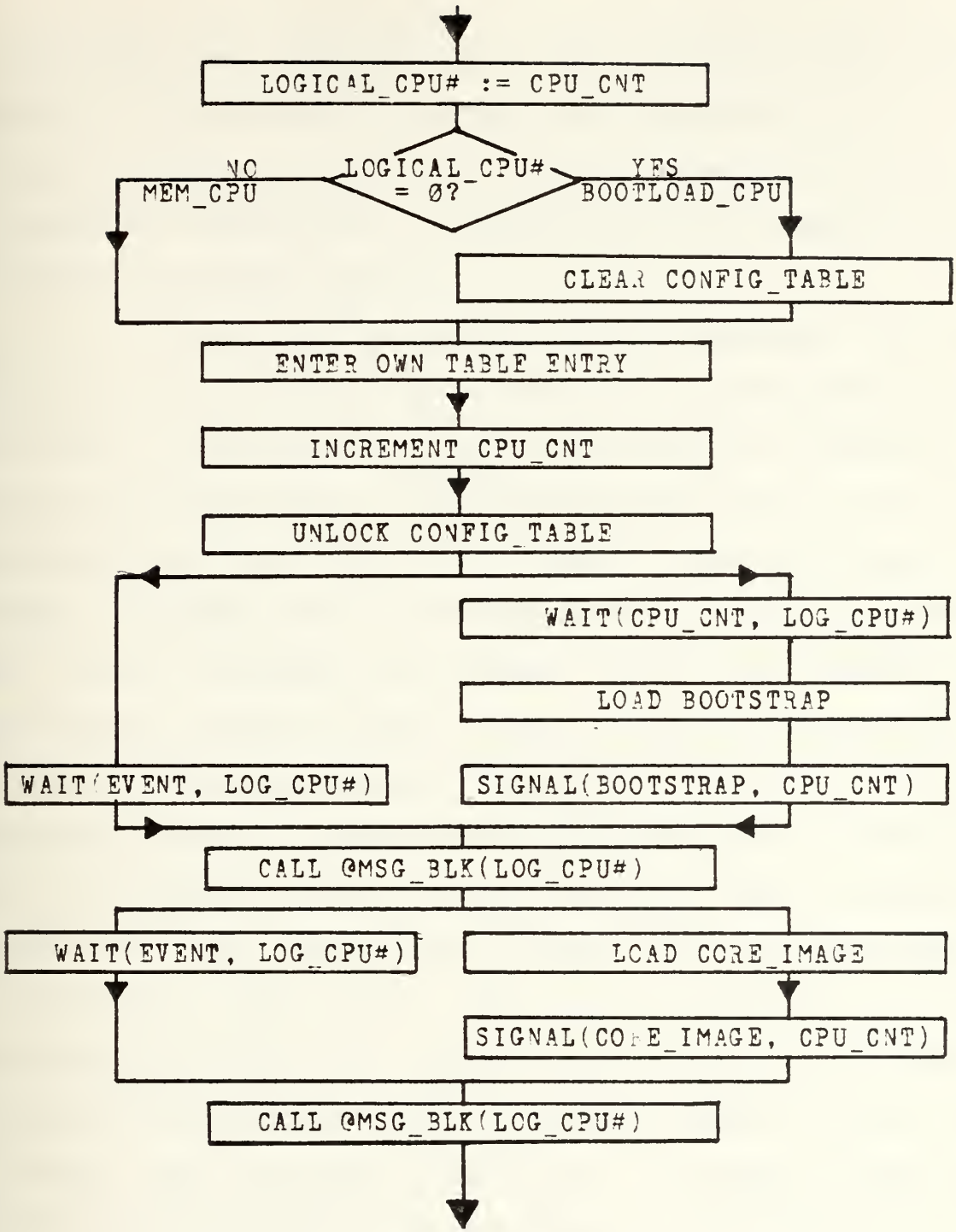
The cooperating processor stage algorithm as shown in Figure II-4 presents the sequence of events for both the bootload CPU and the member CPU's. The bootload and bootstrap programs executed by the processors contain the algorithms for both the bootload and member CPU's. The bootload CPU begins by incrementing the logical CPU number for the use of the next processor to access the table. It then clears the configuration table to accommodate an entry by each processor, enters its own unique ID, maps and enters its own memory map, and then unlocks the configuration table for access by the member CPU's. The memory map is a data structure representation of the local/global determination of each memory block. The bootload CPU then waits (observing

the logical CPU numbers) for all member CPU's to complete their entries before proceeding.

Member CPU's use their logical CPU numbers to index the configuration table CPU list to determine where to place its entry. After entering their own unique ID and memory map, each member CPU increments the logical CPU number indicator, unlocks the configuration table, and waits for an appropriate signal from the bootload CPU to proceed.

The "signal" and "wait" operations performed here are of the type employed by operating system traffic controllers for synchronization between processors, though of a more primitive form. A processor "waits" by looping, looking for the occurrence of an event; a processor "signals" by posting or otherwise establishing the occurrence of an event. The algorithms of Figure II-4 make use of the signal blocks of the configuration table to deterministically signal the passing of information in the message blocks and the granting by the bootload CPU of permission to proceed.

A key issue at this point is the way in which the bootstrap and operating system core images are loaded. In general, each image is loaded into global memory by the bootload CPU; and each member CPU must copy its possibly different, core image into its own local memory. This method of "downloading" the core images allows for the possibility of different types of processors in the architecture, and



Cooperating Processors Stage
Figure II-4

compensates for the fact that the bootload CPU cannot address the local memory of any other processors (i.e., not using dual-ported memory). In the more common homogeneous processor architecture, a shared bootstrap program can be utilized in global memory.

The bootstrap program load is performed by the Bootload CPU interacting with a known (at system generation) secondary storage device. The global memory load address is determined dynamically by the Bootload CPU with use of the configuration table. The entry address of the bootstrap core image is loaded into the message block's of each processor, and all are signalled to proceed. With this method of individual message blocks, each processor may download a different bootstrap program. Transfer of control is effected when each processor obtains the entry point from its message block and executes a call to that location. Each processor passes its own logical CPU number as a "pass by value" parameter in this call, and the location of the configuration table is passed as a "pass by reference" parameter. After the transfer of program control out of the firmware, the bootload program is no longer used. The firmware can if desired, be disabled or disconnected for re-use of the physical address space it occupied.

As can be seen in Figure II-4, the bootload CPU then procedes to determine the address in which to load the operating system core image, while the member CPU's wait for

the appropriate signal to continue. After the core image is loaded, the bootload CPU signals all processors to leave the bootstrap program and enter the base layer of the operating system. Each processors' logical CPU number and the address of the configuration table are again passed into the new program. Once the flow of program control has left the bootstrap program, the memory space that it occupied can be re-used.

In summary, it should be pointed out that nothing was known of the bootstrap program (except its secondary storage address) by the bootload program, or of the operating system core image by the bootstrap program; and the only common link is the knowledge contained in the configuration table which was dynamically determined and passed as a parameter to each successive program.

c. Local Initialization Stage

The local initialization stage is contained within the firmware. The tasks performed by this stage are hardware dependent and include both hardware and software initialization functions specific to each single processor.

Hardware initialization functions include initializing specific purpose internal CPU registers and external devices. Specific purpose registers include pointers and counters used for specific functions on which the hardware depends for normal operation (i.e., PSAP, stack pointer, and REFRESH registers). External devices include

the various input/output interface devices, processor clock/counter devices, auxillary processing components, and interrupt controllers. Each device state or function is determined under software control; the CPU must individually program each device.

Software initialization involves setting the processor's data structures and variables. Examples of these types of data structures include processor interrupt/trap jump vectors and stacks. All tasks performed in this stage are highly implementation and hardware oriented. The local initialization stage in this initialization design contains the program information that must be changed for applications to different hardware architectures.

2. Bootstrap Operations

Bootstrap operations begin after transfer of program control flow from the firmware to the bootstrap core image. Two groupings of tasks are found in the bootstrap program, the global initialization stage and the core image load stage. The global initialization stage performs the hardware resource knowledge consolidation for the system, and the creation of operating system data structures. During the core image load stage the operating system is loaded and a signaled transfer of control occurs which marks the completion of the initialization phase.

a. Global Initialization Stage

The configuration table contains resource information about each individual processor. This information must be interpreted from an integrated system viewpoint in order to establish the system resources. The Bootload CPU consolidates entries in the configuration table to produce the information useful to the base layer of the operating system. The logical-to-physical CPU mappings of the processors is already available, but might be re-arranged into a more convenient format.

Individual memory maps are consolidated into one system map primarily for global memory determination. Consolidation produces a system memory map showing total size and addresses of global memory. The local memory map of each processor has no meaning on the system level except to collectively scope the address space range within the system.

During this stage, the bootload CPU dynamically determines the address location in which to load the core image of the operating system. A pre-allocation entry for the core image is made within the system global memory map.

b. Core Image Load Stage

Loading of the operating system base layer is in general a two-load operation. The Bootload CPU determines a suitable global memory location from the system memory map and loads a portion of the base layer core image. This part

of the operating system includes all processor local code (distributed) and data structures (if pre-established at system generation time). Each member CPU is signaled by the Bootload CPU to "down-load" this core image to an address in local memory determined individually by each processor. The starting address of the core image in global memory and its size are passed in the message block's of the configuration table. Again, individual message blocks allow each processor to possibly have a different core image to download. As each processor completes the task and updates its individual memory map, it increments the CPU count in the configuration table to indicate this. The Bootload CPU is then aware of total task completion; each member CPU again waits for a signal to continue.

The second part of the operating system base layer to be loaded by the Bootload CPU is a global resident core image. The previous core image, having been down-loaded by each processor, can now be over-written in global memory. The Bootload CPU loads the global portion of the base layer core image into an appropriate location in global memory, allocates the locations in the system global memory map, and signals this fact to the member CPU's along with the global location. All processors transfer control to a single base layer entry point, either in local or global memory. This marks the end of the initialization phase and the begining of the run time phase.

D. RUN TIME PHASE

The begining of the run time phase starts with execution of the base layer of the operating system. The run time phase contains an additional initialization stage which primarily serves to format the system configuration knowledge according to operating system specifications, and to virtualize the physical resources prior to normal system operation.

1. Run Time Initialization

Unlike the bootload and bootstrap programs that executed on the bare system hardware, the initialization routines of the operating system base layer can be supported by the operating system functions. But before these support functions are available, the system wide data structures for resource management must be created. The bootload CPU dynamically determines the location of these system tables in global memory from the system memory map, and procedes to build the tables based on information contained in the core image about the structures.

Once the operating system resource management tools have been constructed, the bootload CPU signals the member CPU's to begin execution of the base layer of the operating system. The primitive processor synchronization mechanism used thus far is again used. After each processor begins execution of the base layer of the operating system, the more sophisticated synchronization mechanisms of the

operating system are available, and the bootload CPU distinction is no longer required.

All functions performed during this stage are executed only once; therefore care should be taken to keep the program sections small or to reuse the memory space after execution.

2. Run Time Load

Any additional code or data required from secondary storage to effect the layering of the operating system, can be obtained by use of a loader process within the running operating system, for example, the SASS supervisor for file system initialization. Normal process synchronization services are available to the loader process. A further discussion and example of a loader process has been presented by Anderson[19]. It is interesting to note that the loader process can contain many of the fault tolerant aspects of the system. Once the operating system is running normally the initialization process is completed.

E. SUMMARY

An initialization design for dynamic determination of the resources in an adaptive manner was presented in this chapter. The three phases which sequence the initialization process in time were discussed; the program mediums used to perform the initialization were broken down into functional

stages; and a dynamic parameter passing technique between mediums was presented.

The significant features include the general applicability of the design, the dynamic resource mapping routines, and the processor synchronization method. The general applicability of the design is based on independence of program units, viz., bootload program, bootstrap programs(s), and operating system core image. Dynamic resource mapping is performed on system processors and memory. The processor synchronization method makes use of a global data structure known as the configuration table to facilitate inter-processor communication, and a randomly determined controlling processor for synchronizing processors and interfacing secondary storage.

Chapter IV will apply this initialization design to the SASS Developmental Architecture to implement initialization for this operating system. The next chapter will define the environment in which this initialization design will be implemented. The Secure Archival Storage System will be examined to determine what minimal configuration is required by the base layer; and the hardware architecture built to support the SASS will be studied as part of this minimal configuration.

III. ENVIRONMENT DEFINITION

A. OBJECTIVES

The first consideration in system initialization is the environment definition. As previously stated, system initialization produces a loaded and running operating system. The operating system runs on what it knows as the bare machine or the system configuration. System configuration consists of the software configuration and hardware configuration [17]. The software configuration consists of the values of various system parameters and the sizes of the required data structures or system tables, i.e. the amount of available memory or a bit map of secondary storage. The hardware configuration is defined by the collection of hardware modules comprising the system, and the manner in which they are connected. System initialization is a step-wise evolution from from a fixed (e.g., PROM-resident) bootload program running on the minimum basic hardware to the actual running operating system. A definition of the environment is necessary to effect initialization.

The objective of this chapter is to illustrate this definition process by actually studying a state-of-the-art operating system, the Secure Archival Storage System (SASS), and the assembled developmental architecture on which it

will run. The operating system (SASS) will be discussed in a top-down manner from its desired functionality to its required system configuration. For a more complete analysis the reader is encouraged to consult the referenced literature. The coverage presented in this thesis is meant to familiarize readers with the design structure. The following background is necessary to complete the coverage for all readers. Those already familiar with basic operating system concepts may want to skip to section C. The hardware architecture is covered to the same depth of detail in section D; however, supplementary background information and architectural details are included in reference [27].

B. SECURE OPERATING SYSTEM CONCEPTS

The operating system to be considered is the Secure Archival Storage System which is a subset of a family of secure, multi-microcomputer operating systems described by O'Connell and Richardson [1]. Two primary motivations in the design of this family of secure operating systems were (1) to effectively coordinate the processing power of multiple microprocessors, and (2) to provide information security. Before presenting an overview of SASS, a few fundamental operating system and information security concepts that directly relate to these design motivations should be reviewed.

1. Multiprogramming and Multiprocessing

Fundamental to the concept of multiprogramming is the notion of a process. A process can be described [8] as a set of related procedures and data undergoing execution and manipulation, respectfully, by one of possibly several processors of a computer. It may be considered as a logical entity characterized by an address space and an execution point. A process' address space consists of the set of all memory locations accessible by the process during its execution. Its execution point is the internal state of the processor on which the process is running, at the instance of execution. Both the processor internal state and the running process' address space can be preserved and restored at a later time. This ability to store or re-instate processes on processors is called process switching or context switching.

Multiprogramming is the use of process switching in a manner as to have more than one process in a state of execution at the same time. Asynchronous multiprogramming requires communication between processes for synchronization, e.g., advance and await [16]. Logical attributes of processes include identification (unique ID or processor affinity), classification (interprocess priority or security access class), and state (execution state). For example, in SASS each process is given a unique identifier that allows for its identification by the system. It is also

given a security access class, at the time of its creation, to specify what authorization it possesses. Process execution states allow for multiplexing of processes (binding) onto processors. A process that is bound to a processor is assigned a 'running' state; a process in the 'ready' state is waiting to be bound to a processor; and a process that is 'waiting' is awaiting some event in the system before continuing execution in the 'ready' or 'running' states. Multiprogramming is logically a form of parallel processing, where different processes are in a state of executing simultaneously.

As can be inferred from the preceding paragraph, parallel processing does not require more than one processor. However, in a multiprocessing environment parallel processing can be more effective. Multiprocessing implies more than one central processor in the hardware configuration during system execution.

2. Memory Segmentation

Memory segmentation is a form of memory virtualization. A segment can be defined as a logical grouping of information, such as procedures or data areas, that are of variable length. The address space of each process is comprised of a collection of those segments that can be accessed by that process. Since a segment is a logical unit, it can have logical attributes as does each process. Segmentation thus facilitates enforcement of

controlled access to segments by processes through comparison of certain logical attributes, e.g., access class. Access within a segment is made by two-dimensional addresses: segment specifier and offset.

Segmentation permits multiple processes to share data and code segments, and thus avoid the 'multiple copy' problem. This eliminates the possibility of having conflicting data when multiple copies of the same segment are maintained. Also a reduction in the number of copies reduces the amount of physical address space used. Shared segments greatly facilitate inter-process synchronization and communication for cooperating processes.

3. Abstraction

Dijkstra [15] has shown levels of abstraction to be a powerful design methodology for complex systems. In general, the use of levels of abstraction leads to a better design with greater clarity and fewer errors. Simply put, abstraction is the application of a general solution to a number of specific cases. More precisely, the method of abstraction can be thought of as a methodology for machine virtualization, where each successively lower levels are not aware of the abstractions or resources of any higher levels. Higher levels may apply the resources of lower levels only by making use of the virtual machine provided by the lower level supporting it. These two rules reduce the number of

interactions amoung levels of the system and two rules reduce the number of interactions amoung levels of the system and also contribute to configuration independence in the overall design.

Each level of abstraction creates a virtual machine upon which the next level runs; no knowledge of lower machine implementation is of concern to higher levels. Levels of abstraction can be applied consecutively down to the hardware architecture if desired. Following the rules of abstraction results in a loop free design.

4. Protection Domains

Protection domains are used to arrange process address spaces into rings of different privileges[5]. The structure essentially divides the address space into levels of abstractions with strictly enforced 'ring crossings' or gates. These gates protect the machine hierarchy by enforcing virtual machine boundaries for some, but not necessarily all, levels of abstraction. The inner most ring or domain is commonly considered the most privileged.

5. Kernel Design

Asynchronous operating systems generally fall in two categories: the monolithic operating system approach and the kernel approach. An operating system using the concept of a monolithic operating system is based on the premise that the operating system provides both resource management and the numerous common services required by user programs. User

programs and system devices do not communicate directly; all interaction is passed through an operating system. All functions performed by the operating system are contained in one large program module.

An alternate approach provides a hierarchy of "virtual machines". It is based on a fundamental operating system module called the 'Kernel'. The system's resource management activities are minimized and concentrated in the kernel; various asynchronous activities are moved to asynchronous system processes, as fits the notion of a process as described earlier. These system processes are each given their own virtual processor, which is multiplexed onto the real CPU by the kernel. These virtual processors compete in a multiprogramming sense with other virtual processors onto which application processes are being multiplexed. Kernel activities normally include multiprogramming and application inter-process communication functions.

This smaller module (kernel) is more readily distributed within the address space of each process and is itself constructed as a hierarchy of virtual machines, thereby providing a loop-free, configuration independent structure. The distributed kernel is a key property in the SASS design.

6. Information Security

System security requires the implementation of a security policy from both external and internal aspects [1]. Total reliance on external methods (outside the operating system) precludes the controlled sharing of multilevel information within a system; total reliance on internal methods (provided by operating system) leaves the physical media unprotected. Clearly, systems should employ a combination of the two.

A typical security policy consists of discretionary and non-discretionary aspects. Non-discretionary security is the partitioning of entities into separate, but related classifications. It can best be related to information security through the abstraction of the reference monitor [1]. The reference monitor is composed of subjects, objects, and an access matrix. From an operating system standpoint, the notion of a process generally fits the abstraction as a subject; and data or programs coorespond to objects that can be accessed by subjects. The access matrix represents the permitted accesses between subjects and objects; each matrix contains a lattice structure [13] that defines the relationship between different access classes. The relationship between subjects' access class (SCA) and objects' access class (OAC) is as follows:

1. SAC = OAC read/write permitted
2. SAC > OAC read only permitted
3. SAC < OAC no access permitted

This lattice structure can be partially ordered (not all classes related) or totally ordered (all classes related) as in the typical DOD classification of secret, confidential, or unclassified.

Discretionary security is subserviant to non-discretionary security in that the later dominates any security interpretations. Discretionary security allows for separate, commonly thought of as internal, partitioning within the non-discretionary lattice structure. A typical example is the DOD 'need to know' policy where controlled access is granted or revoked within the non-discretionary policy.

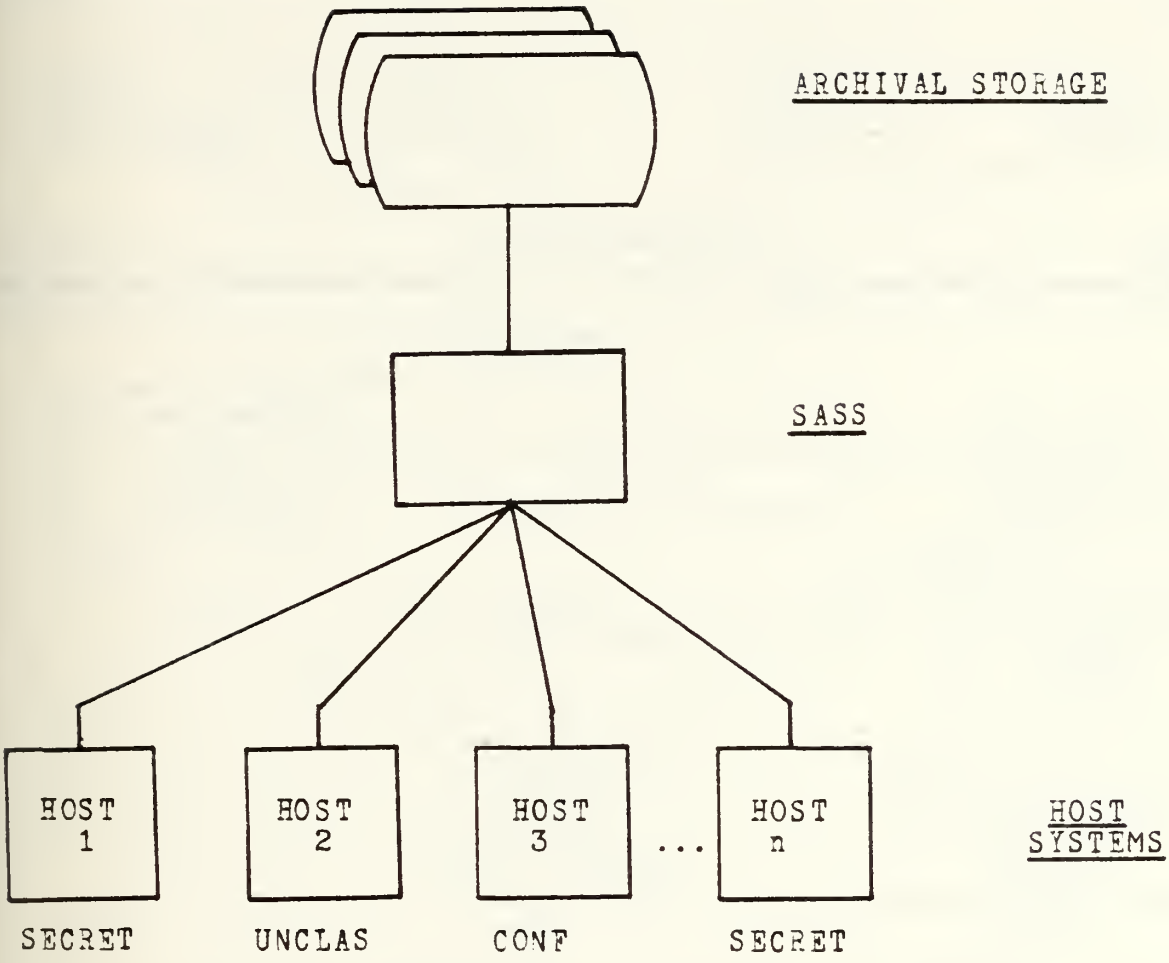
C. THE SECURE ARCHIVAL STORAGE SYSTEM

As stated earlier, the SASS is a member of the family of secure operating systems designed by O'Connell and Richardson [1]. Functionally, it was designed to provide multiple host computer systems with controlled, shared access to a multilevel secure archival storage. This requirement leads to the design goals of internal security to protect information flow in a distributed computer network, configuration independence for both system versatility and security support, and general subsetting

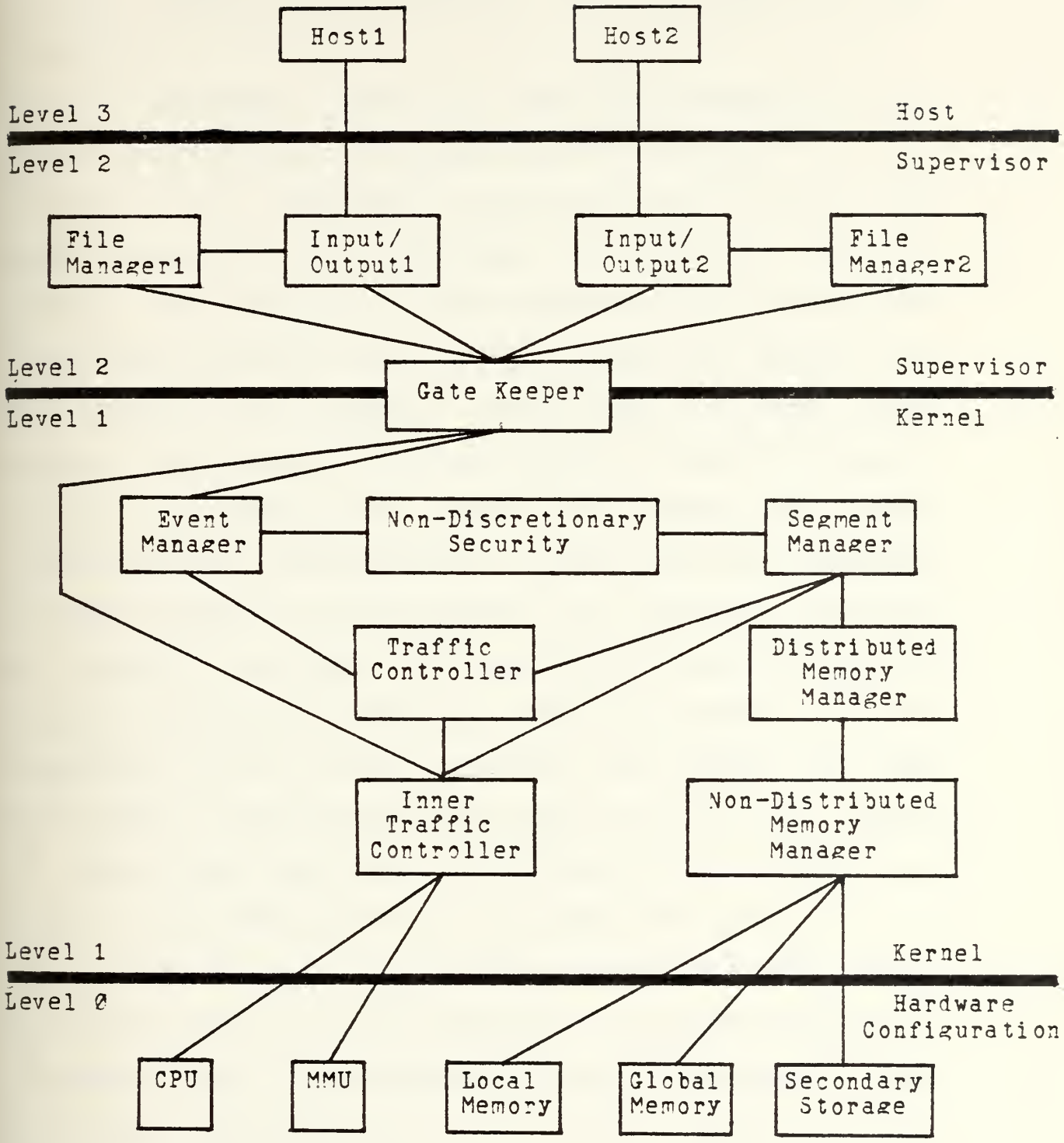
capability to support future system configurations. A general SASS organization can be seen in Figure II-1.

The key elements in the SASS design are the use of security kernel technology [12], a distributed operating system, and resource virtualization. The security kernel is a kernel based design which includes a realization of the abstraction of the reference monitor. Non-discretionary security policy enforcement is added to the basic kernel functions of providing segmentation, multiprogramming and interprocess communication. The system is distributed both logically and physically; logically, parts of the operating system are distributed within the address space of each host system. The use of protection domains permits the operating system to maintain its integrity while interacting with the Host system[1]. Physical distribution of parts of the system, when used in conjunction with shared data bases, eliminates the dependence on a single controlling unit (master CPU). In addition, an increase in performance may be realized by reducing potential bus contention when executing common code.

The SASS environment can now be defined by examination of the four levels of abstraction shown in Figure II-2. Level 3 is the Host computer systems; all that is known by a Host system about the next lower level is the virtual machine interface provided, specifically a set of five instructions: create, delete, read, store and modify files.



SASS Design
Figure III-1



SASS System Overview
Figure III-2

The Host systems thus interface SASS at level 2, the Supervisor.

The Supervisor (level 2) marks the begining of the operating system code. The operating system consists of two domains: the Supervisor and the secure Kernel. The Supervisor operates in the less privileged of the two domains. The function of the Supervisor is to manage the input/output protocol with the Host systems and maintain the hierarchical file structure established for each Host system. Two processes, the I/O process and the File Manager process, are created for each Host system at system initialization. Communications protocol and data packaging is accomplished by the I/O process. All commands received and actions initiated are coordinated by the File Manager process. Functions provided by the File Manager include management of the Host's virtual file system and the enforcement of the discretionary security policy. It should be noted at this point that both levels (2 and 3) exist in a total virtual environment: all resources are virtual.

At the interface between the Supervisor (level 2) and the Kernel (level 1) is the Gatekeeper. All that is known of the lower levels to the Supervisor is a virtual machine with an extended instruction set. The virtual machine is the restricted subset for hardware instructions and the extended instruction set facilitated by the Gatekeeper. The primary objective of the Gatekeeper is to isolate the Kernel and

make it tamperproof [7]. The gatekeeper establishes the logical boundary between the Supervisor and the Kernel. As a matter of course it provides a single software entry point (enforced by hardware) into the Kernel (level 1).

Level 1, the security kernel, consists of two components: the distributed kernel, which logically resides in the address space of each Host system; and the non-distributed kernel. The distributed kernel consists of the Segment Manager, the Event Manager, the Non-discretionary security module, the Traffic Controller, the Inner Traffic Controller and the distributed Memory Manager module. Two modules, the Event Manager and the Segment Manager comprise the extended instruction set contained within the gatekeeper and available to the supervisor. The Segment Manager provides segmented virtual storage management, and the Event Manager provides inter-process communication.

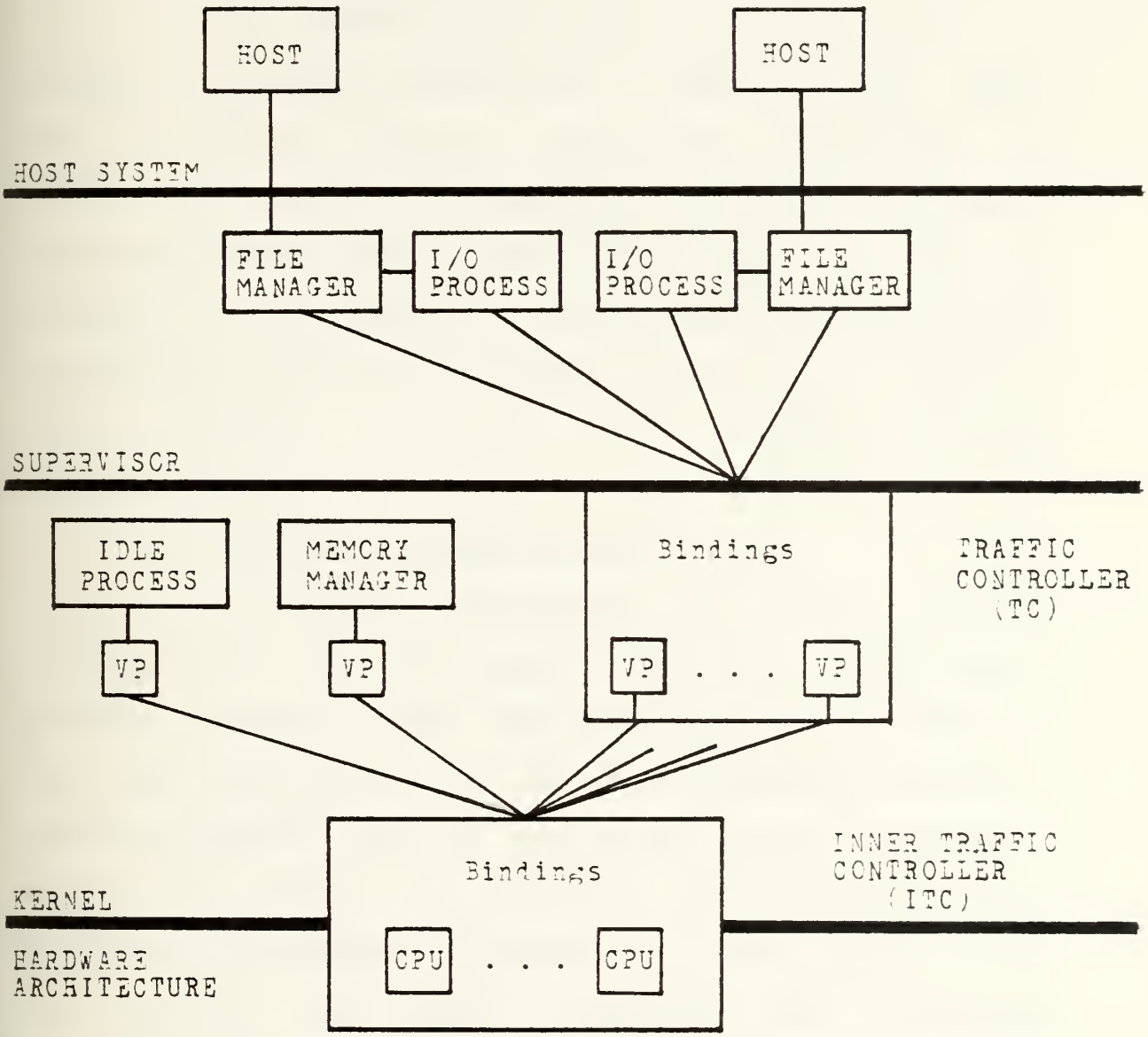
The Traffic Controller is really comprised of two modules: the traffic controller listed above, and the Inner Traffic Controller module. Binding or "mapping" of Host processes to virtual processors is accomplished by the Traffic Controller; binding of virtual processors to real processors is the function of the Inner Traffic Controller. This two-level traffic controller supports the loop-free module hierarchy for a general kernel-based operating system design (see earlier section on kernels). It also supports

use of local memory since the Inner Traffic Controller manages the virtual processors of each separate CPU.

The Non-discretionary security module enforces the non-discretionary security policy, as the name implies. The remaining module, the distributed Memory Manager, facilitates inter-virtual processor communications for synchronization between the Segment Manager (virtual storage management) and the non-distributed Memory Manager that provides storage virtualization of local/global memory and secondary storage.

The non-distributed kernel consists solely of the non-distributed Memory Manager just discussed. This Memory Manager exists as a kernel process (an operating system function placed outside the distributed kernel, as described earlier), permanently bound to a virtual processor and in competition for the physical processor resources managed by the Inner Traffic Controller. Figure II-3 shows the two-level traffic controller design and the vehicles for resource virtualization (both processor and storage).

Level Ø is the system configuration on which the kernel runs. It consists of the full hardware configuration and the data structures describing this environment. A discussion of the full hardware configuration is presented in the following section. The data structures describing the environment must contain complete information about the physical resources to be managed by the kernel, and in a



SASS Two-Level Traffic Controller
Figure III-3

compatible format. A further discussion of the system configuration will be presented at the end of this chapter.

D. HARDWARE ARCHITECTURE

1. Hardware Requirements

Within the SASS design goals as defined by [5], the concept of resource virtualization is the key to the design goals of internal security, configuration independence, and a functional sub-setting capability. Resource virtualization separates higher levels of the SASS design from the bare hardware in such a way as to permit implementation of these design goals. Hardware requirements must then be based on those hardware features that support resource virtualization.

a. Processor Virtualization

A virtual processor is the software representation of a processor that may be functionally different from the actual, physical processor upon which it will run. Processor virtualization also defines a number of logical processors that are data structures that contain a complete description of processes at a certain point of execution on the physical processor. In the instances where the physical and virtual processors are functionally identical, virtualization serves only to multiplex processes (multiprogramming). In either case, hardware requirements to support processor virtualization are those architectural

features that can be used to bind virtual processors to physical processors, in a state of execution, viz., for context switching.

b. Memory Virtualization

Memory virtualization requires the management of primary and secondary physical memory resources to create the illusion of a primary memory which is independent of the actual physical primary memory. This illusionary memory is called the virtual storage. The logical, relocatable, information objects created by memory segmentation, provide an essential memory multiplexing mechanism for the efficient implementation of virtual storage [5]. Memory segmentation also provides a convenient mechanism by which address spaces may be defined in the creation of processes. Hardware architectural features that provide for memory segmentation and support efficient memory virtualization in a multiprogramming environmentare desirable in the SASS design.

c. Protection Domains

A key concept for the implementation of the internal security design goal is protection domains. Protection domains arrange process address spaces into rings of different execution domains, exhibiting a hierarchical layering of privileges. In the virtual processor sense, the execution points of processes become more restricted in less privileged domains. The hierarchical ring structure supports

processor virtualization where processes within each successive ring run on a virtual processor that is more restricted than its base machine.

This ring structure is an hierarchy in which the most privileged domain is the innermost ring. The structure divides the address space into levels of abstraction with strictly enforced gates at the ring boundaries [5]. Protection rings may be created in software, but an hardware implementation, where gate use is enforced by hardware, is much more efficient [14]. Hardware features that restrict "privileged" instruction set usage within the physical processor support a two domain ring structure and thus, can efficiently serve to implement protected domains.

2. Hardware Selection

The hardware architectural features described above - processor and process multiplexing support, a memory segmentation capability, multiple domain memory partitioning, and multiple domain instruction set - are essential to an efficient implementation of the SASS. The Zilog Z8000 family of 16-bit microprocessors with an architecture which supports memory segmentation and two-domain operation was selected as the target machine because of its robust support facilities and close match to design requirements. Further, it was selected because of its commercial availability as an off-the-shelf, single board package for multiprocessor applications and Multibus (INTEL

trademark) compatability. The later feature was deemed necessary to accommodate a more flexible range of peripherals.

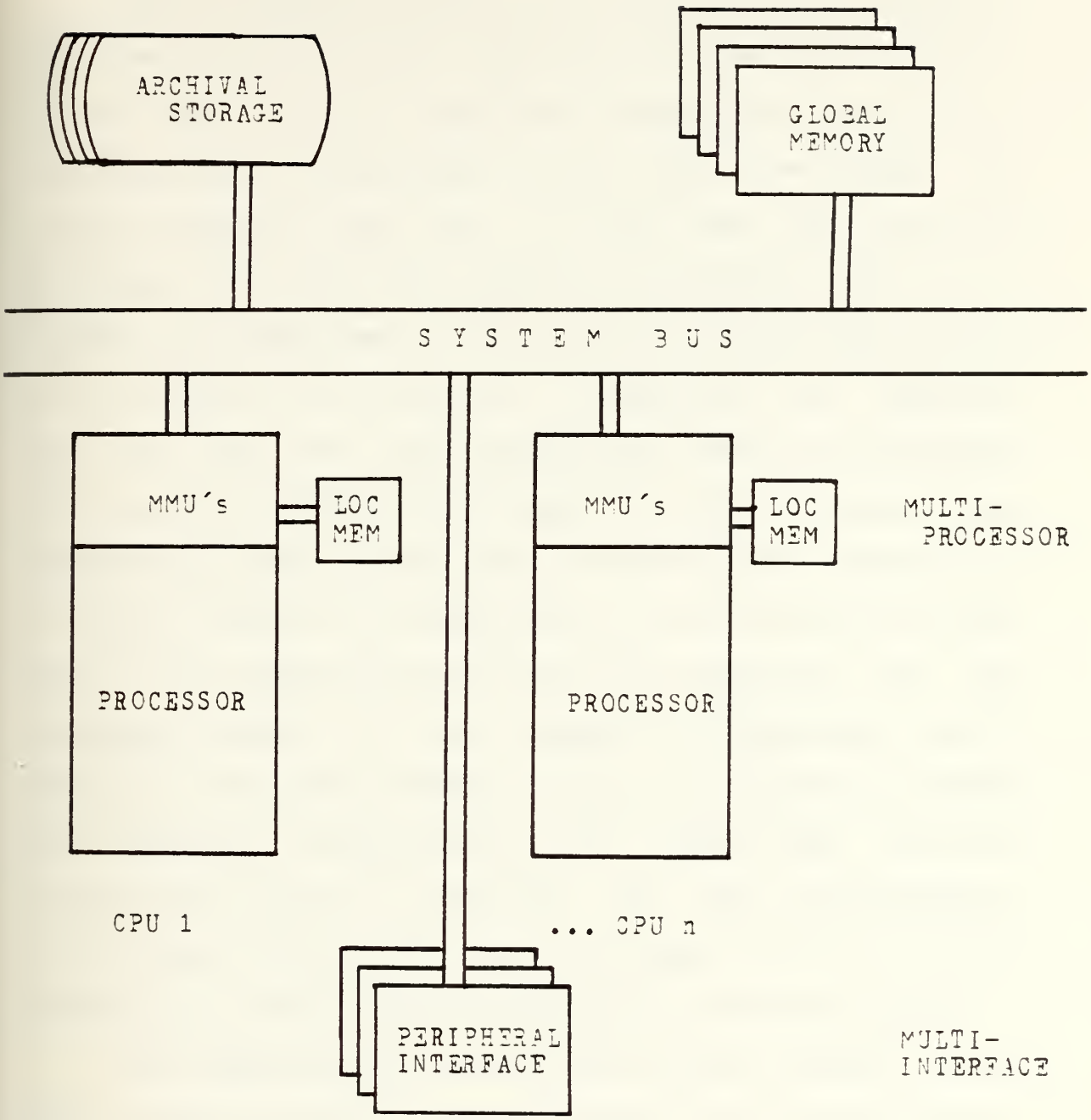
The segmented Z8001 microprocessor is a register oriented machine, with sixteen 16-bit general purpose registers, seven data types (from bits to 32-bit long words), and eight user selectable addressing modes. With the use of the Zilog Z8010 Memory Management Unit (MMU), it can directly access 8 megabytes of memory. A more detailed description of both can be found in reference [27]. The Z8001 hardware was not available for use during system development, and unfortunately, neither was a commercially available, Z8001 single board packaging for SASS architecture implementation. The actual hardware used in the developmental system implementation was the Advanced Micro Computers Am96/4116 Monoboard Computer with the non-segmented AmZ8002 microprocessor.

The Am96/4116 Monoboard Computer provides necesary processor virtualization features. With its Multibus (INTEL) interfacing a processor-to-processor interrupt capability through the Multibus is available. This interrupt becomes the non-vectored interrupt (NVI) source to the Z8002. Local and global time-slicing capability with interrupt is provided by onboard clock circuitry, that also maintains a continuous real-time clock. The above features are useful for effective multiprocessing synchronization.

The Multibus also allows for global memory in a multiprocessor environment. Global memory separation in blocks of 32K bytes is facilitated by five of the 20-address lines provided by the bus and decoded processor signals. Local memory separation is effected by onboard wiring additions (see appendix A) which allow system mode only access to a portion of local onboard memory. Alterations were accomplished with the use of available onboard gates and soldered connections. This memory partitioning affords both kernel code protection and secure process switching in this two domain environment. The more general nature of memory segmentation design has been preserved by software 'simulation' of the MMU hardware.

3. SASS Developmental Architecture

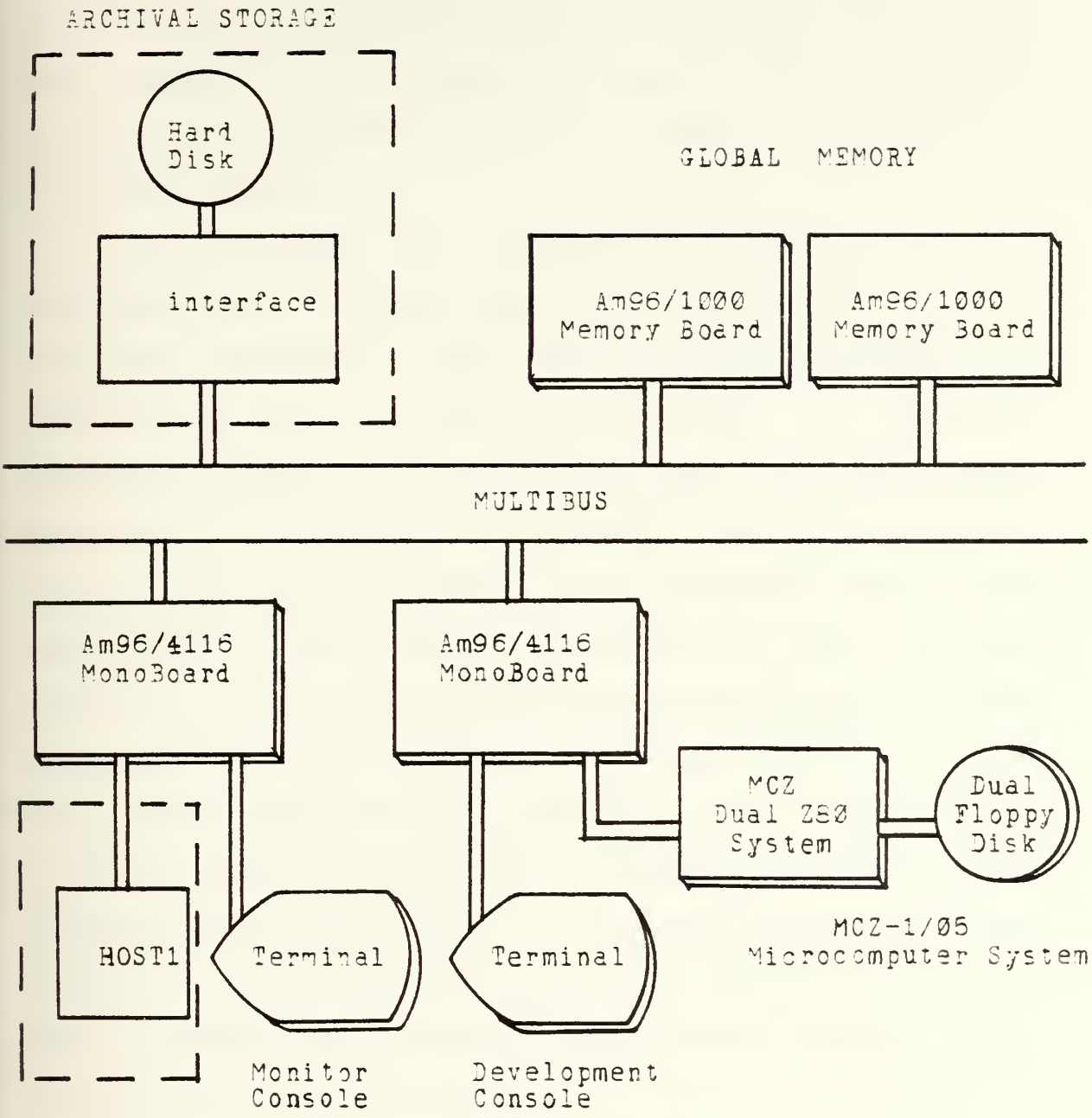
The general hardware architecture of the SASS consists of multiple processors, each with its own local memory and two memory management units, a single system bus, global memory, addressable by all processors, multiple peripheral interface capability, and a large archival storage aggregate. As shown in Figure II-4, an arbitrary number of Z8000 processors and peripheral interfaces (hosts) are assumed, with all components sharing a common system bus. This general architecture is in keeping with the SASS design goals and the characteristics of the Z8000 microprocessor family.



SASS Hardware Architecture
Figure III-4

As a prelude to the actual initialization mechanism development, and to support ongoing and concurrent research on the SASS project, the final hardware architecture was integrated with the existing Zilog Z8000 development MCZ system hardware, that has to date been the principle development tool. The Zilog Z8000 Developmental Module (DM), a Z8000 based experimental board at the heart of the system, has no provision for system bus interfacing and therefore does not meet the SASS design goals. The DM was replaced with the Am96/4116 Monoboard which does have the system bus interfacing capability and facilitates the SASS design goals of global memory, multiprocessors, and archival storage. Once the new single board system was integrated with the remaining portion of the original developmental system, namely the Zilog MCZ-1/05 Z80 based microcomputer system, the physical implementation of the SASS hardware architecture could be realized. The SASS Developmental Architecture evolved as shown in Figure II-5. A detailed description of the system can be found in reference [27].

Two Am96/4116 Monoboards are currently housed in an INTEL ICS-80 chassis, which provides the power supply, cooling fans, and the Multibus backplanes. Each Monoboard is wire-wrapped to specifications listed in reference [27] and integrated into the architecture as shown in Figure II-5. In order to reduce the complexity in physical memory addressing for the non-segmented Z8002, global and local memory sizes



SASS Developmental Architecture
Figure III-5

are assumed equal and indeed implemented in this manner on the Monoboards. The Monoboard assumes a 32K byte block of local memory and a 32K byte block of offboard memory, with its 16-bit address bus.

To reiterate, the local 32K RAM onboard memory has been configured for system mode only access from absolute addresses 0000-3FFF HEX; addresses 4000-7FFF HEX are accessible by both normal and system modes. In addition, attempted access of memory below 4000 HEX while in the normal mode will generate an onboard vectored interrupt. Global memory is addressed from 8000-FFFF HEX on the Am96/1000 RAM Memory Boards interfaced to the Multibus. Currently, three blocks of 32K are utilized and partitioned as normal mode only, system mode only, and normal mode/read only accesses. The read only access RAM must be supported by a direct memory access (DMA) capability in the archival storage device, which has not to date been implemented. DMA capability allows secondary storage to write to primary memory directly, thus providing the necessary access to load the processor read-only memory.

Assembly language programming (PLZ/ASM) for the Z8000 is provided by the Zilog MCZ system software support and an Upload/Download capability with one of the Monoboards. Program development functions are provided by the SASS Developmental Monitor; a complete program listing and command syntax description can be found in reference

[27]. Basically the SASS monitor operates in one of three modes: the transparent mode, in which the Z8000 processor acts in a relay capacity between the MCZ system and the development terminal; the Upload/Download mode, which passes developed Z8000 programs from the MCZ system to SASS system memory; and the typical monitor mode, for program running and debugging. Additional features to support initialization are included and will be discussed later.

Archival storage devices can be almost any available technology, e.g., magnetic fixed or floppy disk systems, optical disk systems, or magnetic tape. Storage capacity, interfacing and timing are the basic considerations of device selection for mass storage within the SASS implementation. As previously mentioned, no storage device has to date been implemented.

An additional architectural feature not previously discussed or shown in the SASS design is the ability to communicate directly with the SASS, not as a peripheral or host system, but as the SASS operator. Communications, such as occur during system initialization, must occur through a terminal device and some program transfer media. An archival storage device may fulfill the later requirement provided external portability exists (viz., a disk that can be removed and transported). This link would most probably be effected through a dedicated peripheral interface. A single parallel port is provided on the same Monoboard for the as

yet, undefined Host system. Future work on the developmental Host interfacing will use this architectural feature.

E. SUMMARY

With the system environment adequately defined, the initialization goals can be identified. These goals appear to fall into two categories: (1) hardware initialization leading to a fully functional hardware configuration; and (2) initialization of the full system configuration on which the operating system will run.

Processor initialization in the SASS architecture requires that a program-status area (PSA) be established, properly loaded, and the PSA pointer register loaded to point to that area. Secondly, a stack area must be allocated and the default (normal/system) stack pointers set appropriately. Thirdly, all necessary interrupt/trap service routines must be made available and identified in the PSA; initialization of external devices must be performed; and the appropriate interrupt structure enabled. Initialization of the full hardware architecture involves establishing an environment of co-operating processors where a system-wide knowledge of physical resources is known.

Initialization of the full system configuration or "bare-machine" on which the SASS will run, consists of making available to the operating system the knowledge of the hardware configuration, which includes all physical

resources and any pre-allocations such as the PSA and stacks. Resource knowledge must be consolidated into a form suitable for resource management. The manner in which these initialization goals are accomplished is the subject of the next chapter.

IV. DESIGN IMPLEMENTATION

This chapter describes the implementation of the initialization design for the SASS Developmental architecture. First, a discussion of the implementation objectives is presented, followed by an explanation of those restrictions peculiar to this effort which are imposed due to either hardware constraints or circumstances. A description of each of the program components follows next. Further, the Bootload program and the Bootstrap program (contained in reference[27]) are examined in detail. Afterwards, a discussion of how these programs relate to the operating system core image when considering run time initialization, is presented.

A. OBJECTIVES

The primary objective of this implementation is to use the design methodology presented in this thesis to effect an initialization mechanism that will produce a running SASS. The implementation must be able to initialize the current demonstration package as well as the final version of the SASS, while working within the developmental architecture. A clear choice between using the developmental environment or running the SASS must be given to the operator. Using the developmental system requires the initialization of a

monitor that provides the operator with the developmental tools he needs. Currently, running the SASS involves running the demonstration package since the operating system implementation is not completed.

A secondary objective requires that the design methodology produce flexible and versatile bootload and bootstrap program structures and algorithms for general application. The bootload program, which comprises the firmware, must be readily adaptable to many hardware architectures. The bootstrap program should accomodate a wide range of monitor or kernel based operating systems. In addition, each component must be independent of the others except for the dynamic parameter passing mechanism.

Since the current SASS demonstration package is not yet functional in a multiprocessor environment, a separate test program must be created to demonstrate the initialization mechanism with multiple microprocessors. These programs will be loaded and run in place of the operating system.

B. RESTRICTIONS

Several hardware related constraints are placed on the implementation. One restriction, which significantly affects dynamic resource determination, is the requirement for Memory Management Unit (MMU) simulation. The use of wiring modifications for local memory protection and hardware domain signals to partition global memory has complicated

the experimental method of determining memory resources. In a hardware supported segmentation environment using MMU's, this is accomplished in a straight forward manner by a read/write determination of the full physical address space; domain signals are not used for memory segregation. However, in the current hardware architecture using domain signals to partition memory, memory resources must be experimentally determined in both the system and normal modes.

Execution of the code for mapping memory in the normal mode must reside in a portion of memory accessible in the normal mode. This means if the code being executed prior to switching to the normal mode is currently in an area accessible only in the system mode, it must be moved to an area addressable in the normal mode. Since a more logical choice would be to execute any code which contains mode switching in an area accessible by both modes, the implementation choice was made to "fix" the location of any code moves at system generation time. This restriction is strictly an implementation dependent detail derived from a simulation environment to start with, and will be discussed further under the bootstrap program section.

A major hardware restriction effecting the overall appearance of the programs but not their functions, is the requirement to use the MCZ microcomputer development system as the secondary storage device to contain the bootstrap and operating systems core images. Effecting a bootstrap loading

of these core images is significantly more involved than utilizing disk controller (i.e., hard disk) primitive operations.

The MCZ ZILOG RIO operating system makes use of a packet-passing protocol (Tectronix format) [24] in order to effect upload/download operations through its serial port connected to the MonoBoard. This packet-passing protocol requires more coding than typical disk controller primitives. A suitable secondary storage device was not available during the development of this implementation.

This increase in the amount of code also led to another implementation choice. While it is true that a developmental monitor program is actually just another operating system, the conventional manner of storing the program is as a part of the firmware. ROM size restrictions when coupled with the increased amount of coding mentioned above, and desire to adhere to the design methodology, led to storing the developmental monitor on secondary storage (i.e., MCZ).

It should be noted that the secondary storage interfacing programs reside within the firmware, and as such are a hardware dependent feature. They must be added to the bootload program when adapting it to a specific hardware architecture.

One software restriction which pertains has been discussed earlier. The source code should be programmed without any absolute addressing. Some form of relative

addressing or base index addressing should be used. This allows the code to be placed at any location in addressable memory and to execute properly. This holds for the bootload program as well, which may reside in a PROM that may be "hardwired" to any physical address. Programs and program data structure core images must then be created to run at absolute address zero to facilitate base indexed addressing.

The resultant implementation choices led to generation of the bootload program for firmware, the bootstrap program for bootstrapping the SASS, and the monitor program for the SASS Developmental Monitor. A discussion of the monitor program is contained in reference [27].

Figures IV-1 and IV-2 provide an overview of the bootload program control flow. Figure IV-3 overviews the bootstrap program. In this implementation the bootload program is composed of two main modules and three support modules. The support modules serve only to interface secondary storage. The bootstrap program consists of one main module and the same three support modules.

C. BOOTLOAD PROGRAM

The system initialization mechanism was designed to commence operating once power is applied to the system, or as is the case with the RESET switch, power is interrupted momentarily. This in turn causes each processor to acquire its initial execution point from within its initial address

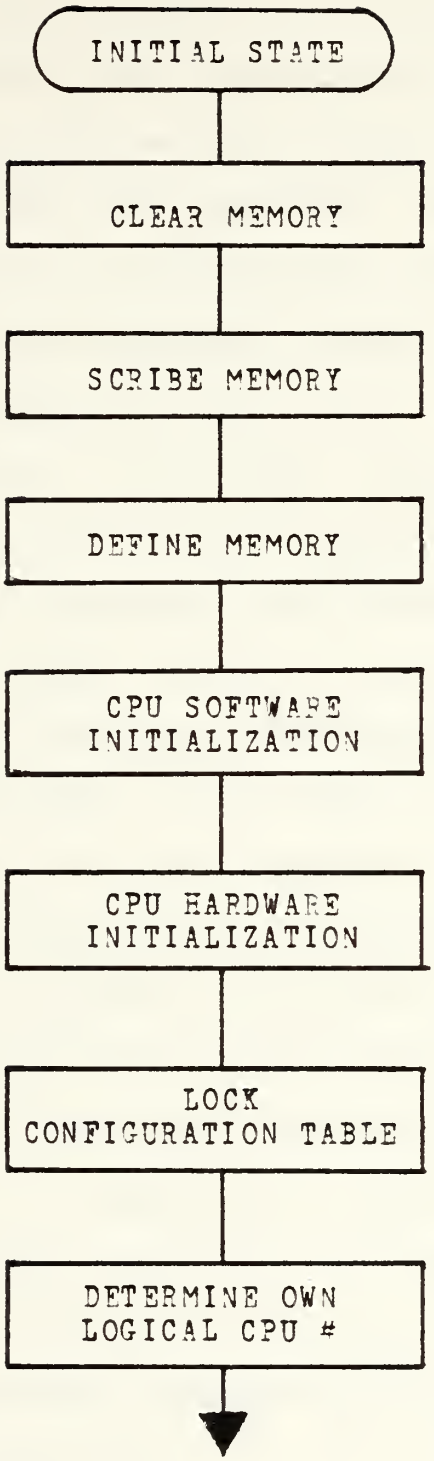
space in the firmware. This address space contains the bootload program. Each processor will have its own serialized firmware, as stated previously. For the Z8000 microprocessor the Flag Control Word (FCW) portion of the initial execution state is obtained from address 0002 HEX and the Program Counter (PC) from address 0004 HEX. Bootload program execution will begin with this defined processor state.

At this point the processor is working in an environment without RAM and with no knowledge of any other processors. This marks the start of the Independent Processor Stage. The bootload program listing is contained in reference[27]. Figure IV-1 shows the independent processor and local initialization stages that comprise the first bootload module.

1. Independent Processor Stage

The tasks of the independent processor stage consists of "clearing" memory and defining primary storage. It has been found in this implementation that the scribing operation of the cooperating processors' tasks can be more efficiently performed concurrently with the clearing and defining operations.

Each processor begins by clearing memory in the system mode by writing a pattern (55AA) at the begining of each block of memory and writing zeros to the next five locations. This read/write pattern was selected for reasons



Bootload1 Module
Figure IV-1

described in chapter II. The size of the block was chosen to be 800 HEX which is the smallest size of ROM supported by the Am96/4116 MBC hardware architecture. Using this method, a routine traverses the physical address space clearing memory. After this is accomplished, each processor waits approximately 2 milliseconds, enough time for any other processors to complete the same task.

Next memory is scribed (SCRIBE_MEM) by each processor by use of the bus locking mechanism. At the conclusion, each processor again waits for any other processors that might exist to complete the task. Each processor is still operating independently.

The DEFINE_MEM routine makes use of the results from the clearing and scribing operations to determine the addresses of the lowest blocks of local and global memory. The results of the scribing operation distinguishes local and global memory by establishing access to memory by more than one processor. Additionally, the highest scribed value obtained during the searching of memory becomes the number of intercommunicating processors in the system. For the purposes of this implementation, the cooperating processors and local initialization stages will rely on the contents of these dedicated registers:

R5 = number of processors
R6 = highest local address
R7 = lowest global address

Notice that all code execution has been sequential (viz., no calls or returns) thus far due to the undefined stack. At this point, available read/write memory for the system mode is known. For the single processor case however, global memory is undefined and must be "set" statically. This can either be done by absolute addressing, as in this implementation (8000 HEX), or by adding an appropriate value to the low local address, to achieve separation for establishing the configuration table. For ease of coding, it is desirable to initialize the internal CPU register used as a stack pointer in order to facilitate procedure calls. It is for this reason and the desire to keep related functions contiguous, that the local initialization stage is accomplished next.

2. Local Initialization Stage

Having an address of accessible memory, certain internal, special purpose CPU registers can be set. These registers are used as pointers into this addressable primary memory space to establish the system mode stack and the program status area (refer to reference [27] for a discussion). The stack pointer is implicitly assigned to register R15 for the non-segmented Z8000. Setting R15 to the stack area contained within the first 100 HEX of the low local address block, facilitates the use of CALL and RETURN instructions. Since all programs produced at system generation time can be located at any physical address, a

dedicated CPU register (R12) must be set to the base address of the current program location to allow for the base indexed addressing mode that is used to acheive relocatability. All procedure and label addresses used during stack operations and control branching are derived from offsets applied to this dedicated code address register.

A similar method is applied to variable references within the code since the address area containing these variables is defined dynamically. The data structures used by the various programs are created as templates at system generation time at absolute address zero, and base indexing with a dedicated data address register (R14) is used in referencing the variables. Once this register is set, the processor is no longer in a variable free environment.

A Z8000 required data structure is its effective interrupt/trap jump vector, or its program status area. Now having available memory, the program status area can be established by setting the program status area pointer (PSAP).

In the current implementation the high block of local memory addressed by register R6 is allocated for the stack, the program status area, and the data area (variables). The stack area is allocated from 0700-07FF HEX with the stack pointer (R15) set to 07F0 HEX relative to R6; the PSA is assigned from 0600-0700 HEX with the PSAP set to

address 0600 HEX relative to R6; the CODE_AREA register R12 is set dynamically to the base of the bootload program location; and the DATA_AREA register R14 which facilitates the use of variables, is set to address 0200 HEX relative to R6.

Included in the local initialization stage is the actual initialization of the single processor data structures. This is collectively called software initialization. The data area pointed to by R14 is cleared; variable storage areas are brought to a known state; and those variables requiring initial values are appropriately set. To enable input/output communications with the console and with the MCZ system (if connected) for upload/download, the input ring buffer must be filled with spaces (blanks).

The PSA is initialized to enable interrupts for communications through the terminal or with the MCZ system. All FCW's for the PSA are initialized to the system mode (4000) to disable additional vectored interrupts until the current handler routine is completed. The console port and MCZ port interrupt handlers (procedures CONINT and MCZEND) are set in the PSA to allow communications; and the non-maskable interrupt (NMI) is set as a way to return from the transparent mode when initializing the MCZ system. In the transparent mode, the bootload program loops, passing data between the console and the MCZ system. In short, those

handlers initially set in the PSA are to enable that I/O which facilitates system initialization.

Those single processor tasks which initalize hardware devices external to the CPU are grouped into the HDW_INIT routine. Components initialized here include those hardware devices without which no I/O would be physically possible. For example, the Interrupt Controller (8259A) chip on the MonoBoard and the serial port USARTs (9551) which service the console and MCZ systems are initialized first. As a matter of convenience, other hardware components though not necessary to continue initialization, are initialized in the HDW_INIT routine as well. In this implementation the Timing Controller (9513) IC is also initialized to provide the clocks, counters, and software interrupt sources to be used later by the system.

The last and probably most important task of the local initialization stage is to physically permit the I/O communications that have been provided for, by enabling the vectored interrupts (i.e., console and MCZ ports). When finished with the local initialization stage the independent processor has a data area for variable storage, an interrupt driven I/O communication mechanism, and all hardware in an initialized state.

3. Cooperating Processor Stage

Actually some of the tasks that comprise the cooperating processor stage were performed concurrently with

those of the independent processor stage as described before. They began with the first locking of the system bus during the scribing operation. At this point in the bootload program, the number of processors and the existence of local and global primary memory is known. The range of the local and global memories is yet to be determined. This is accomplished by mapping memory. Each processor must map its own known memory space in a coordinated fashion and provide this information to the system.

To allow for inter-processor communication the configuration table is implicitly assigned in the low global memory block as pointed to by register R7. As can be seen by the configuration table data structure in Figure IV-2, the read/write pattern location and CPU count used during the scribing operation, for both the normal and system modes, are incorporated into the table. This allows for preservation of the clearing and scribing operations already performed. The configuration table proper begins with the table lock.

The table lock provides the mutual exclusion mechanism for controlled sharing of the table. The next word location is the CPU count which is to be used by each processor to determine its logical CPU number, and by the bootload CPU to "count" processors' responses. All individual processors' entries in the table are contained in the CPU list entry.

```
CONFIGURATION TABLE   RECORD [
          R/W_PATTERN     WORD
          CPU_NUM         WORD
          NORM_R/W_PAT    WORD
          NORM_CPU_CNT    WORD
          TABLE_LOCK      WORD
          CPU_CNT         WORD
          CPU_LIST        ENTRY ARRAY ]

ENTRY ARRAY       ARRAY [ MAX_CPU  CPU_ENTRY ]

CPU_ENTRY                RECORD [
          SIGNAL          WORD
          CPU_ID          WORD
          MSG_BLK         MESSAGE
          MEM_MAP         MEM ARRAY ]
```

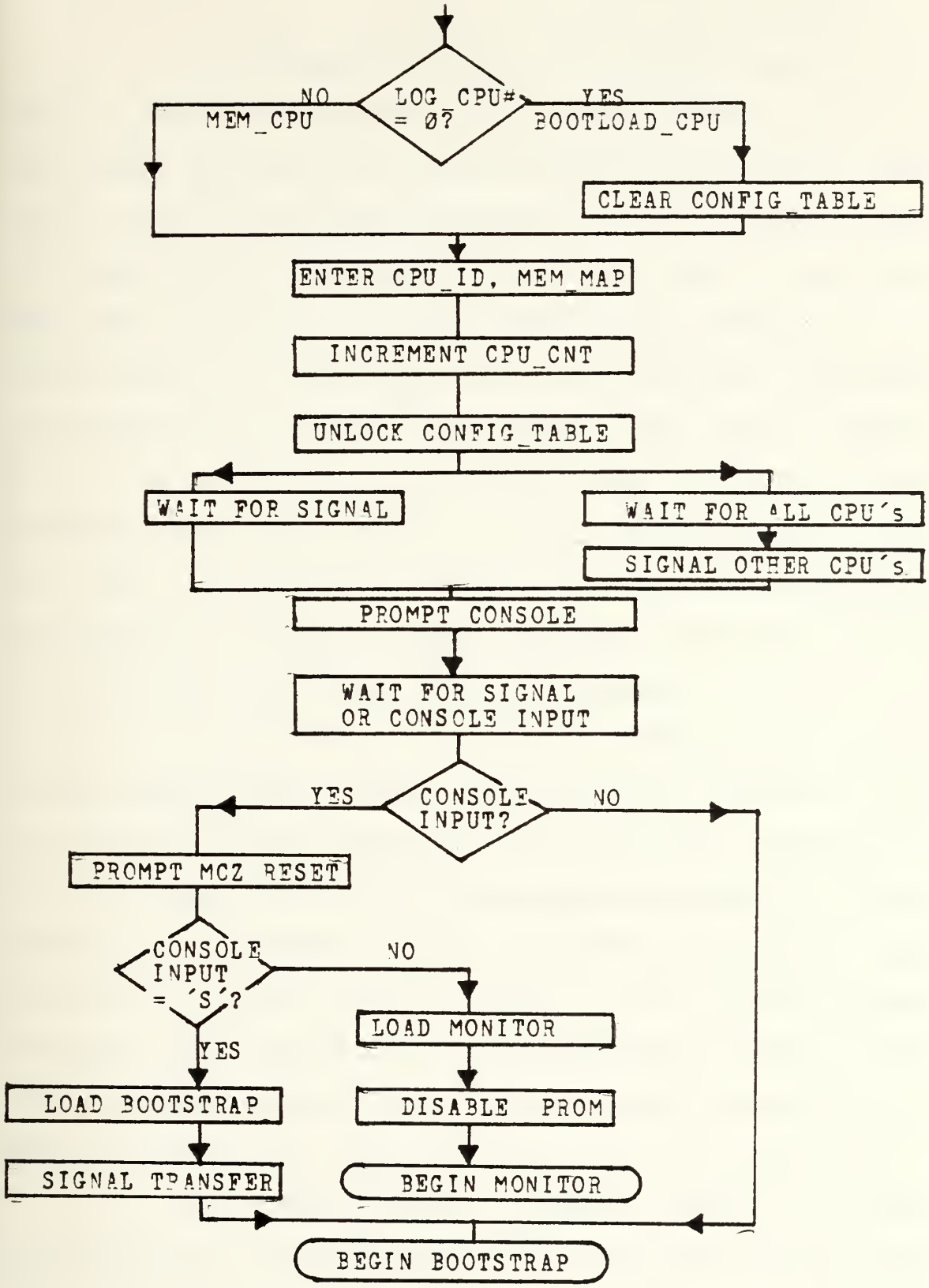
Configuration Table Declaration
Figure IV-2

A bootload CPU must now be determined. Each processor attempts to gain access to the table by setting the table lock. To do this each processor performs a test-and-set operation utilizing the bus locking mechanism to insure the integrity of the operation. The first attempt to lock the system bus by each processor create a race condition, the winner of which becomes the first processor to access the table; and thereby becomes logical CPU Ø and the bootload CPU.

After gaining access to the configuration table, each processor checks to see what its logical CPU number is by the CPU count entry. If it is Ø then the processor becomes the bootload CPU; otherwise the processor becomes a member CPU. A differentiation of processors has now been made and each branches to the appropriate section of code. Figure IV-3 contains the algorithm for the cooperating processors stage that comprises the second bootload module.

a. Bootload CPU

The bootload CPU increments the CPU count to indicate the next processor's logical CPU number. Next it will bring the CPU list entry in the table to a known state by clearing enough CPU entry blocks to accomodate all known processors (R5). After clearing the table entries, the bootload CPU procedes to make its own entry in the table as logical CPU Ø.



Bootload2 Module
Figure IV-3

A CPU entry consists of a signal word, a CPU ID word, a three word message block, and the actual memory map. The signal is used by the bootload CPU to inform the member CPU's that it has placed a message in their message block's and that the next sequential action can now be performed. The CPU ID entry is where each processor enters its own unique identification number. The memory map is a "byte map" of memory blocks for both the system and normal domains. Note that the size of the CPU entry is fixed at system generation time when the block size is determined. The size of the configuration table however, is dynamically determined at runtime. An upper bound of the size of the table is fixed for programming convenience.

The bootload CPU after having entered its own unique identification number into the table, procedes to map its physical address space for the system mode, making use of the "clearing" and scribing results from before. Memory mapping is performed by the MAP_MEMORY procedure which constructs the map with a call to the system mapping procedure. Each memory block is mapped with a word; the high byte represents accessibility in the system mode and the low byte represents the normal mode that will be mapped later.

The system mapping procedure makes an access determination for each memory block. If a given memory block is accessible, as indicated by the presence of the read/write pattern (keep in mind that a read only access is

defined by an instruction fetch operation and not a data fetch as is performed here), it is then designated according to the number of processors having access, as indicated by the scribe location entry. If only one processor has access, as indicated by scribe location equal to one, the block is designated as local memory in the map by a '01' entry. If scribe location is equal to the total number of processors known to the system, the memory block is labeled as global ('02'). Access by a number of processors greater than one but less than the total, defines the block as non-usable ('04'). All memory blocks not containing the read/write pattern are not accessible and designated '05'. The '03' designation will be used later during the normal mode mapping operation to indicate access in both the normal and system modes.

After having made its own memory map entry into the table, the bootload CPU unlocks the table to allow access by the other processors. It then waits for all member CPU's to make their entries; this is indicated by the CPU count which is incremented by each processor after it has made its entry in the table.

b. Member CPU

Each member CPU procedes to obtain its logical CPU number from the CPU count and to compute the base address of its entry. It will next make its own entry: first its unique identification number and then its own memory

map. The member CPU then increments the table CPU count and unlocks the table to allow access by the remaining processors.

Each member CPU will now wait for the signal from the bootload CPU to continue. A slight delay between signal checks is added to reduce contention for the system bus while making global memory accesses.

c. Bootstrap Loading

The normal sequence of events within the bootload program would be to have the bootload CPU load the bootstrap program and then signal a transfer of control by all processors, out of firmware and into the bootstrap program. However, in this implementation, only one processor can be connected to the MCZ microcomputer system being used for secondary storage. To preserve the generality of the design, the bootload CPU in this implementation was not 'forced' to be that processor attached to the MCZ system. It is therefore necessary at this point to determine which processor is connected to the MCZ and allow that processor to effect the bootstrap program download and transfer of control by all CPU's.

An initial prompt (*) is sent to the console port of each processor to signify that bootload operations have been completed (except for bootstrap loading) and that I/O is now established. Each processor then enters a program loop, waiting for either an input from the console or a

signal word entry in the configuration table. An input from the console designates the processor attached to the MCZ system. A signal word entry signals a transfer of program control to the bootstrap program entry point contained in the first word of the message block.

The processor which is attached to the MCZ system becomes the bootload coordinator to effect the download of a program from the MCZ. The bootload coordinator first insures that the MCZ system is initialized by sending the prompt 'RESET MCZ' to the console for operator action, and entering a transparent mode where all further console entries are relayed to the MCZ system and visa versa. To exit the transparent mode the NMI interrupt that returns program execution to the NMI_RTN point is used.

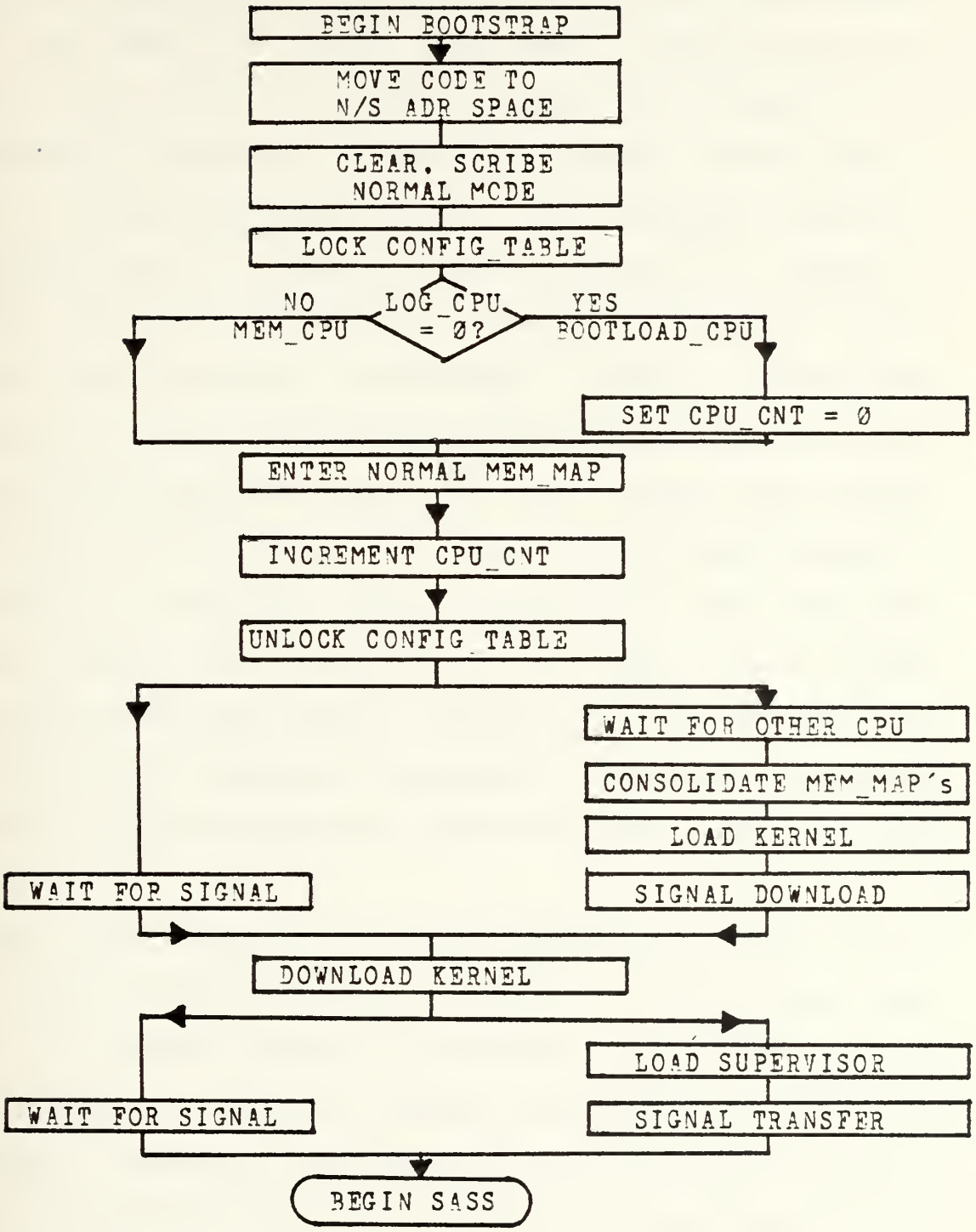
At this point the bootload coordinator must permit the console operator the choice of running the SASS Developmental Monitor or the SASS. An 'S' entered from the console signifies a bootstrapping of SASS, while any other entry denotes the developmental system. In the former case, the bootload coordinator effects the downloading of the bootstrap program; in the latter case, the SASS Developmental Monitor is downloaded. A single processor system is assumed when establishing the developmental system, with the other processors not being loaded with the monitor. The monitor program is written "under" the shadowed firmware and the PROM's must be disabled before execution of

the monitor can begin. A procedure to disable the PROM moves the code outside the firmware into local memory and effects a transfer to the code, that disables the PROM's and transfers to the monitor program entry point. In the case where the operator selects SASS loading, the bootload coordinator loads the program and signals the transfer of control to all other processors by using the signal word and message block entries of each processor.

D. BOOTSTRAP PROGRAM

The bootstrap program performs the hardware resource knowledge consolidation and operating system loading. This is accomplished in two stages: the global initialization and the core image load. In this implementation the complete mapping of memory requires mapping in the normal mode as well. The normal mode mapping could not have been performed at the same time as the system mode mapping since the PSA was undefined at that time. The PSA is required for the use of the Z8000 system call (SC) instruction which facilitates mode switching, more specifically the switching from the normal to the system mode. The SC instruction causes an internal CPU trap to the program status area for determination of the trap handler which is executed in the system mode.

The start of the bootstrap program, as seen in Figure IV-4, performs the normal mode memory mapping. As mentioned



Bootstrap Module
Figure IV-4

earlier in the chapter, the code for the normal mode mapping must be moved to a "fixed" local area of memory accessible both in the normal and system modes. The inability to dynamically determine an area to which the code may be moved, without first executing the code, creates a difficult situation; hence, the code relocation address is fixed at system generation time. This situation is a product of the current architectural restrictions. Given a suitable secondary storage device with DMA capability, and the already existing global read-only (code) memory the dilema no longer exists. The bootstrap program would simply be loaded by the secondary storage device into this read-only global memory and execution of the code could proceed sequencially in both domains without any relocations. Of course with the segmented architecture memory mapping within domains is not applicable and the code may be removed entirely.

The code is moved to a common system/normal mode area of local memory and control is transferred. In the normal mode (less privileged domain), I/O instructions and instructions the change CPU special purpose registers can not be executed; therefore the system call instruction is used to effect execution of certain of these privileged instructions while in the normal mode. In particular, the bus lock and unlock instructions (I/O) and the switching of modes (changing the FCW register) are required. The first section

of code 'clears' and scribes memory in the same manner as used for the system mode.

At this point the bootload CPU must again be determined. Each processor attempts to gain access to the configuration table and when successful, compares its own logical CPU number (passed as a parameter from the bootload program) with the CPU count. If it finds a match, a check for logical CPU number Ø (bootload CPU) is made; the bootload CPU resumes its role and the member CPU's resume their roles. In the same manner as for the system mode, the bootload CPU maps and enters its own normal mode memory accesses. It then supervises the same task for the remaining member CPU's. Again the mapping code must be moved by each processor to the pre-defined local address location (4100 HEX) that is accessible in both the normal and system modes.

1. Global Initialization Stage

The global initialization stage tasks include consolidation of the individual processors' memory maps to form a system map having global memory knowledge and the creation of a logical-to-physical CPU map. In this implementation, no further formatting of the resource knowledge is performed since the processes within the SASS that use the knowledge have not been implemented. In more general applications, this section of the code may contain further processing of this resource knowledge into a "standard" form.

The resource knowledge now available includes: a unique identification for each system logical CPU; a local memory map for each processor; and a system wide global memory map.

2. Core Image Load Stage

The core image load stage involves first loading the kernel portion of the operating system, and then loading the supervisor portion. The bootload CPU loads the Kernel (of SASS) into a available global address that it determines from the system global memory map. After the core image is loaded, the bootload CPU sets the message block of each member CPU to (1) the address where the core image was loaded, (2) the address in local memory to where it is to be moved, and (3) the number of bytes to be moved. The bootload CPU next downloads the core image into its own local memory and signals the other processors to download as well.

The second task of the core image load stage is to load the Supervisor, into an appropriate location in memory. In this implementation the Kernel and Supervisor addresses are fixed at system generation time since the core image contains absolute addressing; future SASS implementations presumably need not contain absolute addresses and the core image load addresses could be determined dynamically at run time.

E. RUN TIME INITIALIZATION

With the current implementation no operating system data structures are initialized within the bootstrap program. Some of these functions are currently performed at the start of execution of the SASS core image. For exmaple, Strickler[7] in his work assembled these functions into one 'bootstrap loader" module which basically initializes the data structures used by the Inner Traffic Controller for processor management. After this level of the operating system is initialized, the next level, the Traffic Controller for process management, is initialized, and so forth in a layered manner. These initalization tasks are based on the minimal configuration assumed by the base layer of the operating system, and as such are more related to the core image of the operating system than to the hardware configuration.

Those initialization tasks that create and intialize operating system databases used for the purposes of resource management, are considered as run time initialization. The resources defined in this operating system include primary and secondary storage, processors, processes, and memory segments. Of these, the ones associated with the minimal configuration are processors and memory. This appears to be generally the case with most operating systems; therefore the code for initializing the data structure containing knowledge of the processors and local/global storage should,

as in this implementation, be included within the bootstrap program. The databases for management of any other resources, such as the Active Process Table or Virtual Processor Table used in SASS, should be initialized during runtime initialization or during the layering of the operating system. In the SASS example, this refers to processes and memory segments.

Unique to SASS is a different resource that is directly managed by process management. This resource is the Host systems. These are known to the hardware architecture through I/O ports and interfaces which are hardware devices. These devices must be initialized before communication can be established with the Host systems. The existence of these ports cannot be as easily "discovered" as were the processors and memory. Each must be initialized prior to any interaction with the processors. Though the number of ports or interfaces is not known at system generation time, the manner in which they are addressed or communicated with is known. A "shotgun" approach to device initialization can be taken throughout this addressing range. A similar action is performed in this implementation during hardware initialization where the MCZ port is initialized by all processors for the possibility that the MCZ system is attached.

Once the lines of communication are open, the Host systems can be communicated with. However, each Host system has an access class assigned to its process abstraction. These must be assigned prior to any communications with the Host systems, and therefore must be associated with the ports. At some point between the initialization of the ports and communication with the Hosts systems, an association of these logical attributes to the appropriate port must be made. Since an upperbound on the number of ports is implicit in the addressing method, a database containing an entry for each possible Host system can be constructed at system generation time within the core image. This table at system generation time would statically show no existing Host systems.

The addition or subtraction of Host systems becomes effective during the initialization phase, primarily in the bootstrap program, when the system resource knowledge is consolidated. It is at this point that the knowledge of memory resources and processor resources is passed to the base layer of the operating system. Any knowledge of Host system additions or deletions should also be passed at this point. Since these changes require operator intervention, a separate "cold-boot" bootstrap program having an interactive code section at this point, must be used to facilitate the changes. The knowledge of these changes, or of no changes, can then be used by the base layer of the operating system

to "update" its Host system database, which resides with the operating system core image on secondary storage, during run time initialization.

F. SUMMARY

A detailed description of the bootload and bootstrap programs which effected initialization of the SASS was presented in this chapter. The implementation design adhered to the general initialization design presented in chapter III, with any differences pointed out. Those sections of the programs having general application were distinguished from the system dependent sections; and the reasons for each were explained.

In addition, a method for the treatment of Host systems as resources was presented and a possible implementation described. The actual implementation is dependent on SASS design issues not yet addressed.

V. CONCLUSIONS

A. SUMMARY OF RESULTS

This thesis has presented a multiprocessor initialization design for dynamic determination of resources in an adaptive manner. The design is general in nature and therefore applicable to a wide range of hardware architectures and operating systems. The three phases of initialization were addressed and two program mediums were identified, the bootload program which comprises the firmware, and the bootstrap program contained on secondary storage. The bootload phase was broken down into groups of tasks or stages: (1) Independent Processor Stage, (2) Cooperating Processor Stage, (3) Local Initialization Stage, (4) Global Initialization Stage, and (5) Core Image Load Stage. The independent and cooperating processor stages and the local initialization stage make up the bootload program which resides in ROM. The remaining stages comprise the bootstrap program.

The independent processor stage dynamically determines the existence of local and global memory and other processors, while operating in a variable free environment (viz., no RAM available). In the cooperating processors stage each processor provides to the system in a coordinated manner, the knowledge of its own unique identification from

firmware and a complete local and global memory map. Each processor completes its own initialization functions in the local initialization stage. These functions include setting the internal special purpose registers, establishing its own CPU data structures, and completing the initialization of its own hardware devices. In the global initialization stage the resource knowledge provided by each processor is consolidated into the system databases which establish the minimal configuration for the operating system base layer. The core image load stage effects the loading of the local and global sections of the operating system and starts them running.

The significant features of the design include the general applicability of the design, the dynamic resource mapping scheme, and the hardware synchronization mechanism. The general nature of the design is based on independence of the program modules, i.e. bootload, bootstrap, and operating system base layer. No knowledge of one is assumed by the other at system generation time, rather the knowledge is dynamically passed between units. Dynamic resource mapping is performed on the processors and primary storage, to support the minimal configuration. The hardware synchronization method makes use of a global database known as the configuration table to facilitate interprocessor communication by a randomly selected controlling processor

'bootload CPU) that synchronizes processors and interfaces secondary storage.

An implementation using this initialization design was accomplished for a member (SASS) of a family of secure, multiprogramming, multi-microcomputer operating systems. The implementation was used to effect a running SASS demonstration module in a single processor environment.

B. FOLLOW ON WORK

SASS initialization as provided by this implementation will have to undergo several modifications before the final version will exist. An effort was made to concentrate those areas requiring future change into one program medium, the bootstrap program. The firmware should require only minor modifications, affecting only the secondary storage interfacing primitives. The bootstrap program must be modified as the evolution of the SASS progresses. Those areas for modification are discussed in chapter IV. No significant follow on work to this implementation is required.

The SASS system provides possible areas in both the hardware architecture and operating system that would be suitable for immediate continued research. In the area of hardware, the selection and interfacing of a suitable secondary storage device to the SASS Developmental Architecture is of most immediate concern. This modification

would facilitate the full hardware architecture realization and would thereby permit further implementable work in the SASS in the area of the Supervisor. Another hardware concern is the manner of Host computer systems interfacing. The Am96/4116 MonoBoard supports two serial and one parallel ports for this use. The use of fiber optics to interconnect Host systems to these ports may be of special interest from a security standpoint as a deterent to signal interception.

In the area of SASS, several areas are of immediate interest. The completion of the run time initialization stage as discussed in chapter IV is required before a multiprocessor version of SASS can be attained. Further work in the Kernel includes the actual implementation of the memory manager process for resource management. The implementation of the Supervisor has not been addressed to date. Its areas of research include the implementation of the File Manager and I/O processes, and the final design and implementation of the SASS-Hosts protocols. Another interesting area could be the use of the idle process to perform some useful work. Some of the functions performed during initialization could again be used as preventative diagnosis by the idle process, to provide a measure of fault tolerance. Other interesting areas include the implementation of dynamic process creation, deletion and loading, and the support of multilevel Host systems.

LIST OF REFERENCES

1. O'Connell, J. S., and Richardson, L. D., Distributed Secure Design for a Multi-Microprocessor Operating System, MS Thesis, Naval Postgraduate School, June 1979.

2. Parks, E. J., The Design of a Secure File Storage System, MS Thesis, Naval Postgraduate School, December 1979.

3. Coleman, A. R., Security Kernel Design for a Microprocessor-Based, Multilevel, Archival Storage System, MS Thesis, Naval Postgraduate School, December 1979.

4. Moore, E. E. and Gary, A. V., The Design and Implemention of the Memory Manager for a Secure Archival Storage System, MS Thesis, Naval Postgraduate School, June 1980.

5. Reitz, S. L., An Implementation of Multiprograming and Process Management for a Security Kernel Operating System, MS Thesis, Naval Postgraduate School, June 1980.

6. Wells, J. T., Implementation of Segment Management for a Secure Archival Storage System, MS Thesis, Naval Postgraduate School, September 1980.

7. Strickler, A.R., Implementation of Process Management for a Secure Archival Storage System, M.S. Thesis, Naval Postgraduate School, March 1981.

8. Organick, E. J., The Multics System: An Examination of Its Structure, MIT Press, 1972.

9. Madnick, S. E., and Donovan, J. J., Operating Systems, McGraw Hill, 1974.

10. Reed, P. D., Processor Multiplexing In a Layered Operating System, MS Thesis, Massachusetts Institute of Technology, MIT LCS/TR-167, 1979.

11. Schell, R.R., Dynamic Reconfiguration in a Modular Computer System, Ph.D. Thesis, Massachusetts Institute of Technology, May 1971.

12. Schell, Lt.Col. R. R., "Security Kernels: A Methodical Design of System Security," USE Technical Papers (Spring Conference, 1979). p. 245-250, March 1979.

13. Denning, D. E., "A Lattice Model of Secure Information Flow," Communications of the ACM, v. 19, p. 236-242, May 1976.

14. Schroeder, M. D., "A Hardware Architecture for Implementing Protection Rings," Communications of the ACM, v. 15, no. 3, p. 157-170, March 1972.

15. Dijkstra, F.W., "The Structure of the 'THE' Multiprogramming System", Communications of the ACM, v. 11, p. 341-346, May 1968.

16. Reed, P. D., and Kanodia, R. K., "Synchronization With Eventcounts and Sequencers," Communications of the ACM, v. 22, no. 2, p. 115-124, February 1979.

17. Luniewski, A., A Simple and Flexible System Initialization Mechanism, M.S. Thesis, Massachusetts Institute of Technology, May 1977.

18. Ross, J.I., Design of a System Initialization Mechanism for a Multiple Microcomputer, M.S. Thesis, Naval Postgraduate School, June 1980.

19. Anderson, R.L., Automatic Recovery in a Real-Time, Distributed Multiple Microprocessor Computer System, M.S. Thesis, Naval Postgraduate School, December 1980.

20. Advance Micro Devices, Am9513 System Timing Controller, Product Specification Sheet, 1980.

21. Zilog, Inc., Z8001 CPU Z8002 CPU, Preliminary Product Specification, March 1979.

22. Zilog, Inc., Z8010 MMU Memory Management Unit, Preliminary Product Specification, October 1979.

23. Advanced Micro Computers, AM96/4116 AmZ8000 16-Bit MonoBoard Computer, User's Manual, 1980.

24. Zilog, Inc., Z8000 PLZ/ASM Assembly Language Programming Manual, 03-3055-01, Revision A, April 1979.

25. Schell, R. R. and Cox, L. A., Secure Archival Storage System, Part I - Design, Naval Postgraduate School, NPS52-80-002, March 1980.

26. Schell, R. R. and Cox, L. A., Secure Archival Storage System, Part II - Segment and Process Management Implementation, Naval Postgraduate School, NPS52-81-001, March 1981.

27. Baker, G. S., Secure Archival Storage System - Hardware Architecture and Developmental Monitor, Naval Postgraduate School, NPS52-81-011, June 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4.	LTCOL Roger R. Schell, Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
5.	Lyle A. Cox, Jr., Code 52C1 Department of Computer Science Naval Postgraduate School Monterey, California 93940	6
6.	Joel Trimble, Code 221 Office of Naval Research 800 North Quincy Arlington, Virginia 22217	1
7.	Department Chairman Department of Computer Science United States Military Academy West Point, New York 10996	1
8.	INTEL Corporation Attn: Mr. Robert Childs Mail Code: SC 4-490 3065 Bowers Avenue Santa Clara, California 95051	1
9.	John P.L. Woodward The MITRE Corporation P.O. Box 208 Bedford, Massachusetts 01730	1

10. Digital Equipment Corporation 1
Attn: Mr. Donald Gaubatz
146 Main Street
ML 3-2/E41
Maynard, Massachusetts 01754

11. Joe Urban 1
University of Southwestern Louisiana
P.O. Box 44330
Lafayette, Louisiana 70504

12. LCDR Gary Baker, Code 37 3
COMRESPATWINGPAC
NAS Moffett Field
Moffett Field, California 94031

13. CPT Anthony R. Strickler 1
Route #12
West Shipley Ferry Road
Kingsport, Tennessee 37663

14. Professor T.F. Tao, Code 62Tv 1
Department of Electrical Engineering
Naval Postgraduate School
Monterey California 93940

15. James P. Anderson 1
Box 42
Fort Washington, PA 19034

16. Director, Central Intelligence Agency 1
Attn: Mr. Charles Kellum
Office of R & D
Washington, D.C. 20505

Thesis
B16855 Baker
c.1 193307
Initialization
design for dynamic
determination of
resources.